# Réalisation d'un compilateur

Référence : chapitre 5 dans les notes de cours.

## Introduction

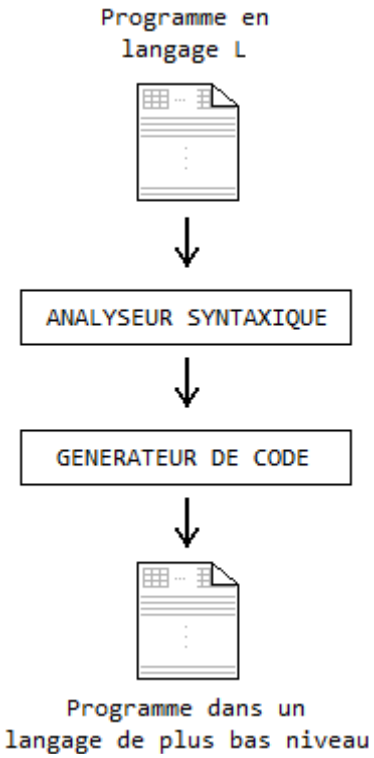
Pour réaliser un compilateur pour le langage L, il faut passer par les 4 étapes suivantes:

1. Avoir une idée précise du langage de programmation L souhaité.

2. Écrire la grammaire G du langage L.

3. Sur base de cette grammaire G, écrire en langage C les fonctions de reconnaissance de programmes écrits en langage L. Ces fonctions forment l'**analyseur syntaxique**. Elles ont pour but de déterminer si tout programme en langage L respecte la grammaire G du langage L ou non.

4. Ajouter dans les fonctions de l'analyseur syntaxique les instructions de génération du code. Le **générateur de code** a pour but de fournir, à partir de tout programme exprimé en langage L, un programme dans un langage de plus bas niveau équivalent.



# Écriture du compilateur

Supposons la grammaire suivante (cf. Chapitre 5 dans les notes de cours) :

```
<plusmoins> → <foisdiv> | <plusmoins> + <foisdiv> | <plusmoins> - <foisdiv>
<foisdiv> → <facteur> | <foisdiv> * <facteur> | <foisdiv> / <facteur>
<facteur> → ( <plusmoins> ) | <nombre>
<nombre> → <nombredeci> | 0b<nombrebin>
<nombredeci> → <chiffredeci> | <nombredeci><chiffredeci>
<chiffredeci> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<nombrebin> → <chiffrebin> | <nombrebin><chiffrebin>
<chiffrebin> → 0 | 1
```

Voici le compilateur construit à partir de cette grammaire. Le code que ce compilateur génère est en langage assembleur.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

void PlusMoins();
void FoisDiv();
void Facteur();
void Nombre();
int  NombreDeci();
int  NombreBin();
int  ChiffreBin();
void CaractereSuivant();
char CaractereCourant();

char acExpr[] = "0b11 * ((3 + 0b101) - 2)";  // expression à compiler
char bCourant = 0;        // indice du caractère courant dans l'expression

void main()
{
   PlusMoins();    // démarrer la compilation

   if(CaractereCourant() != 0)
      printf("Le caractere %c a la position %d dans %s est invalide!",
          acExpr[bCourant], bCourant + 1, acExpr);
   else
      printf("La compilation s'est bien deroulee!");
}

// Cette fonction prend en compte les règles de grammaire:
// <plusmoins> → <foisdiv> | <plusmoins> + <foisdiv> | <plusmoins> - <foisdiv>

void PlusMoins()
{
   FoisDiv();

   while(CaractereCourant() == '+' || CaractereCourant() == '-')
   {
```

```
        switch(CaractereCourant())
        {
            case '+':
                CaractereSuivant();

                FoisDiv();

                printf("pop  ebx\n");
                printf("pop  eax\n");
                printf("add  eax, ebx\n");
                printf("push eax\n");

                break;

            case '-':
                CaractereSuivant();

                FoisDiv();

                printf("pop  ebx\n");
                printf("pop  eax\n");
                printf("sub  eax, ebx\n");
                printf("push eax\n");
        }
    }

    if(CaractereCourant() == 0)
        printf("pop  eax\n");
}

// Cette fonction prend en compte les règles de grammaire:
// <foisdiv> → <facteur> | <foisdiv> * <facteur> | <foisdiv> / <facteur>

void FoisDiv()
{
    Facteur();

    while(CaractereCourant() == '*' || CaractereCourant() == '/')
    {
        switch(CaractereCourant())
        {
            case '*':
                CaractereSuivant();

                Facteur();

                printf("pop  ebx\n");
                printf("pop  eax\n");
                printf("imul eax, ebx\n");
                printf("push eax\n");

                break;

            case '/':
                CaractereSuivant();
```

```
            Facteur();

            printf("pop  ebx\n");
            printf("pop  eax\n");
            printf("mov  edx, 0\n");
            printf("div  ebx\n");
            printf("push eax\n");
        }
    }
}

// Cette fonction prend en compte les règles de grammaire:
// <facteur> → ( <plusmoins> ) | <nombre>

void Facteur()
{
    if(CaractereCourant() == '(')
    {
        CaractereSuivant();

        PlusMoins();

        if(CaractereCourant() == ')')
            CaractereSuivant();
    }
    else
        Nombre();
}

// Cette fonction prend en compte les règles de grammaire:
// <nombre> -> <nombredeci> | 0b<nombrebin>

void Nombre()
{
    if(strncmp(&acExpr[bCourant], "0b", 2) == 0)
    {
        CaractereSuivant();
        CaractereSuivant();

        NombreBin();
    }
    else
        NombreDeci();
}

// Cette fonction prend en compte les règles de grammaire:
// <nombredeci> -> <chiffredeci> | <nombredeci><chiffredeci>
// <Chiffre> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

int NombreDeci()
{
    int iVal = 0;

    if(isdigit(CaractereCourant()) != 0)
    {
        while(isdigit(CaractereCourant()) != 0)
```

```
        {
            iVal = (iVal * 10) + (CaractereCourant() - 0x30);

            CaractereSuivant();
        }

        printf("push %d\n", iVal);
    }

    return iVal;
}

// Cette fonction prend en compte les règles de grammaire:
// <nombrebin> -> <chiffrebin> | <nombrebin><chiffrebin>

int NombreBin()
{
    int iVal = 0;

    if(ChiffreBin() != 0)
    {
        while(ChiffreBin() != 0)
        {
            iVal = (iVal * 2) + (CaractereCourant() - 0x30);

            CaractereSuivant();
        }

        printf("push %d\n", iVal);
    }

    return iVal;
}

// Cette fonction prend en compte les règles de grammaire:
// <chiffrebin> -> 0 | 1

int ChiffreBin()
{
    if(CaractereCourant() == '0' || CaractereCourant() == '1')
        return 1;
    else return 0;
}

// Cette fonction passe au caractère suivant dans l'expression

void CaractereSuivant()
{
    do
        bCourant++;
    while(acExpr[bCourant] == ' ');    // éliminer les espaces
}

// Cette fonction retourne le caractère courant de l'expression

char CaractereCourant()
```

```
{
   return acExpr[bCourant];
}
```

## Étapes de réalisation du compilateur

1. Étendre la grammaire donnée à la page 2 pour permettre également la reconnaissance des nombres en hexadécimal (0x...).

2. Étendre le compilateur donné dans les pages 2, 3 et 4 pour prendre en compte la reconnaissance des nombres en hexadécimal.

3. Étendre la grammaire pour permettre également la reconnaissance des 2 opérateurs : et logique (&) et ou logique (|). Les classes de priorité à respecter sont les mêmes que celles pour le langage C.

4. Étendre le compilateur et tester.

5. Étendre la grammaire pour permettre également la reconnaissance des 3 opérateurs unaires : nombre négatif (-), nombre positif (+) et non logique (~).

6. Étendre le compilateur et tester.

7. Étendre la grammaire pour permettre également la déclaration de variables entières signées sur 4 octets. Le séparateur entre 2 déclarations est la virgule. Une valeur est systématiquement assignée à une variable lors de sa déclaration. Le nom d'une variable est i*x* où i signifie qu'il s'agit d'un entier sur 4 octets et *x* est un chiffre compris entre 0 et 9.

   Par exemple, la grammaire doit accepter le programme suivant:

   ```
   VAR
      i1 = 3,
      i3 = 8
   FVAR
   INSTR
      i1 = i1 * 2
   FINSTR
   ```

   On a à présent 2 parties dans un programme : une partie où sont déclarées les variables (`VAR déclarations FVAR`) et une autre partie où est placée l'instruction d'affectation (`INSTR instruction FINSTR`).

8. Étendre le compilateur et tester.

   Par exemple, à partir du programme donné au point précédent, voici le programme en assembleur que doit générer le compilateur:

```
int i1 = 3;
int i3 = 8;

void main()
{
   _asm
   {
      push i1
      push 2
      pop  ebx
      pop  eax
      imul eax, ebx
      push eax
      pop  i1
   }
}
```

9. Étendre la grammaire pour permettre également l'usage de plusieurs instructions placées les unes après les autres. Le séparateur est la virgule.

   Par exemple, la grammaire doit accepter le programme suivant:

```
VAR
   i1 = 3,
   i3 = 8
FVAR
INSTR
   i1 = i1 * 2,
   i3 = (-i1 * 15) / i3 + 8
FINSTR
```

   On a à présent plusieurs instructions possibles dans la partie `INSTR instructions FINSTR`.

10. Étendre le compilateur et tester.

11. Étendre la grammaire pour permettre également l'usage de variables entières signées sur 1 et sur 2 octets. Le nom d'une variable d'un octet commence par b et le nom d'une variable de 2 octets commence par s. Cette 1re lettre identifiant le type de la variable est suivie par un chiffre compris entre 0 et 9.

    Par exemple, la grammaire doit accepter le programme suivant:

```
VAR
   s1 = 0x3,
   i2 = 5,
   b1 = 0b110
FVAR
INSTR
   s1 = i2 - b1,
   i2 = s1 * 3
FINSTR
```

12. Étendre le compilateur et tester.

13. Mettre à jour le compilateur pour qu'il puisse lire le programme source depuis un fichier et écrire le programme en assembleur dans un autre fichier dont l'extension est *.c*.

14. Étendre la grammaire pour permettre également l'usage de variables entières non signées sur 1, 2 et 4 octets. Le nom d'une variable entière non signée commence par la lettre u, il est suivi par la lettre b, s ou i selon la taille: 1 octet, 2 octets ou 4 octets, de la variable, puis par un chiffre entre 0 et 9.

    Par exemple, la grammaire doit accepter le programme suivant:

```
VAR
   i1 = -3,
   ub2 = 5,
   ui2 = 72
FVAR
INSTR
   ub2 = (ui2 + 3) * i1,
   ui2 = ub2 * ui2 / 2
FINSTR
```

15. Étendre le compilateur et tester.

## Bonus

Pour les étudiants qui ont terminé les étapes précédentes, vous pouvez ajouter à votre compilateur la reconnaissance des flottants à simple précision (type float en C) :

16. Étendre la grammaire pour permettre l'usage de variables flottantes à simple précision. Leur nom commencera par la lettre f, laquelle sera suivie par un chiffre entre 0 et 9.

    Par exemple, la grammaire doit accepter le programme suivant:

```
VAR
   i1 = -3,
   ub2 = 5,
   f1 = 72.3
FVAR
INSTR
   ub2 = f1 + 3 * i1,
   f1 = ub2 * -(i1 / 3)
FINSTR
```

17. Étendre le compilateur et tester. Le compilateur doit être capable de générer des instructions SSE pour les traitements à faire sur les flottants.

## Remarques générales

- Les programmes en assembleur générés par votre compilateur doivent être acceptés sans la moindre erreur par Visual Studio.

- Votre compilateur doit suivre les règles du langage C au niveau de:

  - Les classes de priorité des opérateurs.

  - La règle du langage C suivante: toute valeur d'une taille inférieure à 4 octets doit être étendue à 4 octets avant l'application de l'opérateur, ceci en tenant compte du genre de la valeur (signé/non signé) dont on étend la taille.

- Le travail sera à faire individuellement. Dans le cas où plusieurs étudiants présenteront le même travail le jour de l'évaluation, la note de chaque étudiant sera divisée par le nombre d'étudiants présentant le même travail.