Day 0 – The ROOTZ OF PARSING TEKNICS

Sommaire

Introduction

Avant de commencer et pour les curieux voici une liste de liens qui permet de comprendre *l'état* d'esprit des concepteurs de C et C++.

The Development of the C Language. : *un* bref historique du *développement* du langage C.

Historique de C++ : De *CFront* a C++.

Interview de Dennis Ritchie. : *l'interview* de son concepteur.

Interview de bjarne Stroustrup. : *l'interview* du concepteur de C++.

Interview de Dennis Ritchie, Bjarne stroustrup, et james gosling. : *interview* des concepteurs de C, C++, java et leurs opinions.

Replacez toujours une technologie par rapport à l'époque où elle a été conçue. Le C a un peu plus de 30 ans (1970). Les problèmes techniques rencontrés en 70 sont maintenant mieux connus et bien mieux maîtrisés. De plus les évolutions de celui-ci à travers C++ (ou java, ou objective C) même si elles ont corrigé pas mal des défauts du C en ajoutant de nouveaux concepts, ont aussi ajouté de nouveaux problèmes.

De l'avis même de leur concepteur, aucun des langages présentés ici n'est à 100% de perfection. Le pragmatisme des auteurs de ces 3 langages phares permet de relativiser. Il nous amène à réfléchir. Si les auteurs des 3 principaux langages actuels ont un tel recul, comment ne pas adhérer à leur vision ? Aucun langage n'est parfait, c'est juste une question de point de vue.

Tout cela pour introduire le projet...

Et si nous ajoutions au C ce qui nous a toujours manqué ? Ce qui nous semble pratique ? Ce qui nous ferait plaisir ? Cette question est l'axe principal de ce cours. Toutefois, de cette motivation ludique initiale, nous n'oublierons pas cependant d'aborder un contenu pédagogique plus austère mais essentiel:

- Paradigme des langages (principalement objet)
- Techniques de compilation
- Mécanisme interne et runtime des langages

Le Projet KOOC (Kind Of Objective C)

Le but est d'écrire un sur-ensemble du langage C pour lui ajouter des fonctionnalités. Ces fonctionnalités devraient pallier certains défauts du C vus en cours. Vous aurez pour le projet final toute liberté dans le traitement de nouvelles fonctionnalités. On ne vous demande pas de refaire un vrai compilateur, mais un préprocesseur intelligent. En appliquant votre programme vous obtiendrez un fichier en C particulier + un (des) fichier(s) de soutien, contenant une part de code original et une part de code généré automatiquement.

La Méthode

Dans un premier temps nous étudierons un outil de *parsing* & de génération de code qui nous permettra de réaliser le *préprocesseur*. Ensuite, nous ajouterons de nouveaux mots-clefs et structure de langage au C afin de l'améliorer et le faire évoluer pour supporter les principes de la POO (programmation orientée objet).

Mais pas de bol aujourd'hui pas d'outil, c'est :

Comment faire un "parseur" à la main plutôt qu'avec les pieds !!!

Sommaire:

- Parsing à la main parce qu'il le faut bien
- Exemple guidée : strncmp

- BNF ou comment spécifier un parseur
- La récursivité de la mort
- Exercices

1) Parsing à la main parce qu'il le faut bien

Afin de comprendre l'outil qui nous permettra de réaliser KOOC, nous allons commencer par simuler à la main (en C++) son comportement. C'est-à-dire que nous allons apprendre à concevoir et coder un "parseur" *from* scratch.

Pour ceux qui ne le savent toujours pas un "parseur", c'est un programme qui :

1. *Lit du texte à partir d'une source de données quelconque (chaîne de caractères passé en paramètre, fichier, connexion réseau).*
2. *Identifie dans le texte lu un certain nombre de formes (caractères ou flux de bits).*
3. *Valide le contenu lu par rapport à certain enchaînement de formes (règles ou grammaire).*
4. *Fournit (ou communique) avec d'autres programmes un certain résultat sous forme de structures de données ou de messages.*

Comme la liste précédente le montre, un parseur ne travaille pas forcément avec un texte en entrée. Il peut aussi travailler avec des données binaires lues d'une connexion réseau. Sous cette définition, on peut y mettre la majorité des programmes de traitement de données. D'où l'importance des techniques présentées ici.

Le point central est : Comment identifier les formes et valider leurs séquences?

C'est-à-dire les étapes 2 et 3. Les étapes 1 et 4 sont moins générales et dépendent du contexte dans lequel tourne le parseur et du but à atteindre.

Pour cela, nous allons réfléchir à partir d'un algorithme simple et connu : **strncmp**.

2) Exemple guidée : strncmp

Ce qui est notable sur la fonction **strncmp** par rapport à la version simple **strcmp**, c'est qu'elle permet de tester si une chaîne est au **début** d'une autre plus grande. On dit qu'elle fait une lecture **"cadrée à gauche"**, car elle ne se préoccupe que du début de la chaîne.

Exemple en C:

```
     const char  *s1 = "bonjour";
     const char  *s2 = "chaussure";
     const char  *s3 = "bonjour, je suis un texte long";

     if (strcmp(s2, s1) == 0) //non
     ...
     if (strcmp(s3, s1) == 0) // non
     ...
     if (strncmp(s2, s1, strlen(s1)) // non
     ...
     if (strncmp(s3, s1, strlen(s1))) // oui
     {     //nous avons lue s1 dans s3
           s3 += strlen(s1);// nous passons a la suite
           ... //
     }
```

Ainsi avec une fonction simple nous pouvons découper facilement un texte lu sur l'entrée standard par rapport à une liste de mot-clef.

Exemple en C:

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

int         main()
{
  char            text[4096];

  // on lit des données
  fgets(text, 4095, stdin);

  // voilà une liste de mots clefs
  char            *keywords[] = {
    "if",
    "while",
    "else",
    "{",
    "}",
    "(",
    ")",
    "commande1",
    "commande2",
    "commande3",
    "commande4",
    "commande5",
    ";",
    " ",
    "\n",
    NULL
  };

  char      *flux = (char*) text;
  while (*flux != '\0')
    {
      int   idx;
```

```
      for (idx = 0; keywords[idx]; ++idx)
      {
        // si on a trouvé
        if (strncmp(flux, keywords[idx], strlen(keywords[idx])) == 0)
          {
            // on affiche le mot clef
            printf("K:%s\n", keywords[idx]);
            // on passe au mot suivant
            flux = flux + strlen(keywords[idx]);
            break;
          }
      }
      // si je suis a la fin, c'est que je n'ai pas trouvé
      if (keywords[idx] == NULL)
      {
        printf("Ne connait pas : <%s>", flux);
        break;
      }
    }
}
```

Ce genre d'algorithme simple, à plusieurs particularités :

- Lui aussi est cadré à gauche
- Il consomme au fur et à mesure le flux
- On gère les séparateurs au même niveau que les mots-clefs

De manière générale, on s'attache à conserver l'ordre des mots au fur et à mesure qu'ils sont identifiés. Nous remplissons la deuxième étape du *parsing*. Nous avons identifié les formes, reste à reconnaître l'enchaînement de ces formes à travers des règles logiques.

3) BNF ou comment spécifier un parseur

Nous en arrivons au problème proprement dit de "représentation de la logique du flux d'entrée". Comment spécifier l'ordre et les répétitions des formes que nous avons identifiées? Pour cela nous allons utiliser un système de notation standardisé appelé **EBNF** pour **Extended Backus Naur Form**. Evolution des premiers systèmes de notation mis au point par John Backus (fortran) et Paul Naur lors de l'implémentation d'ALGOL.

Grâce à une syntaxe simple, nous décrivons sous forme de suite de règles le flux à valider.

Syntaxe de la BNF:

```
::= signifie "est définie par"
A décrit une règle lorsqu'il est avant ::=
A appelle une règle lorsqu'il est après ::=, on parle aussi de clause
"ABC" est un littéral de type chaîne de caractères qui vaut ABC
'A' est un littéral de type caractère qui vaut A
```

```
'a'..'z' est un caractère compris entre les lettres 'a' et 'z'. Les
      caractères 'a' et 'z' respectivement inclues.
->'a' lit les caractères jusqu'au caractère 'a'
| est un séparateur d'expression signifiant "OU ALORS" entre plusieurs clauses
[ ] autour d'une expression (ensemble de clause) définit un groupe
      Les groupes peuvent être répétés par l'ajout des caractères suivants :
      ? pour un groupe optionnel (0 ou 1 fois)
      + pour un groupe qui se répète (1 ou n fois)
      * pour un groupe optionnel qui se répète (0 ou n fois)
      begin..end pour un groupe qui se répète begin fois au minimum,
      Jusqu'a end fois maximum (begin et end deux expressions entières)
; termine une règle
```

Nous pouvons par exemple reprendre le premier programme qui découpe les mots clefs sous forme EBNF:

```
MonFlux ::= [ "if"
    | "while"
    | "else"
    | "{"
    | "}"
    | "("
    | ")"
    | "commande1"
    | "commande2"
    | "commande3"
    | "commande4"
    | "commande5"
    | ";"
    | " "
    | "\n"
]+
;
```

Ici nous spécifions une règle "*MonFlux*" qui se contente d'identifier 1 ou N mot-clef de la liste (entre crochet séparés par OU ALORS) consécutivement.

ATTENTION : le séparateur d'expression | qui signifie OU ALORS est souvent appelé alternative. C'est un opérateur prioritaire sur l'espace qui sépare les clauses entres elles.

Ainsi:

```
R ::= A B C
      | D E F
;
```

veut dire:

Lit la clause A, suivie de la clause B suivie de la clause C, OU lit la clause D, suivie de la clause E, suivie de la F. On ne doit pas lire A,B, puis C OU D, E et F.

Mais où est la valeur ajoutée ? En d'autre terme, quels sont les apports de cette méthode ?

1. Les règles BNF permettent d'avoir l'abstraction (l'algorithme) de ce que fait le code C...
2. Un code C de parsing peut être spécifié en BNF.
3. Et réciproquement, des règles BNF peuvent être converties en C.

En fait, nous pouvons créer autant de règles que nous souhaitons, et appeler dans la définition d'une règle autant de règles existantes.

Ce qui logiquement permet d'écrire des choses comme ceci:

```
MonFlux ::= [UnIf | UnWhile]+
;

UnIf ::= "if" "(" Commandes ")" [ UnePhrase | UnBlock ]
;

UnWhile ::= "while" "(" Commandes ")" [ UnePhrase | UnBlock ]
;

UnePhrase ::= Commandes ";"
;

UnBlock ::= "{" [UnePhrase]+ "}"
;

Commandes ::= "commande1" | "commande2" | "commande3" | "commande4" |
"commande5"
;
```

Ceci commence à ressembler à un petit langage ! Mais comment cela marche-t-il ? Nous décrivons de la règle la plus générale, jusqu'aux règles les plus spécifiques, le flux que nous souhaitons valider. Grâce à la signification de chaque symbole, nous pouvons lire en français le contenu d'une règle.

Par exemple:

```
MonFlux ::= [UnIf | UnWhile]+;
```

Peut se lire approximativement:

La règle ***MonFlux*** est définie par un groupe répétant 1 ou N fois la lecture de la règle ***UnIf*** ou de la règle ***UnWhile***.

Voilà pour l'essentiel le fonctionnements de la BNF, mais comment ceci se code en C/C++?

4) La récursivité de la mort

Nous allons voir maintenant une technique simple pour traduire une grammaire écrite en BNF en un programme. Le type de parseur obtenu est appelé "parseur en descente récursive". Sans rentrer plus en détail dans la théorie des langages, on parle de parseur LL pour *Left Lookahead*.

Il existe d'autres implémentations de parseur LL plus optimales, mais celle présentée dans ce cours a le mérite d'être simple et rapidement maîtrisable.

Cela consiste en la traduction systématique de chaque règle par une fonction ***booléenne*** récursive équivalente.

par exemple:

```
Commandes ::= "commande1" | "commande2" | "commande3" | "commande4" |
"commande5"
;
```

se traduit simplement ainsi :

```
int         Commandes(char **flux)
{
      if (strncmp(*flux, "commande1", strlen("commande1")) == 0)
      {
            *flux = *flux + strlen("commande1");
            return 1;
      }
      else if (strncmp(*flux, "commande2", strlen("commande2")) == 0)
      {
            *flux = *flux + strlen("commande2");
            return 1;
      }
      else if (strncmp(*flux, "commande3", strlen("commande3")) == 0)
      {
            *flux = *flux + strlen("commande3");
            return 1;
      }
      else if (strncmp(*flux, "commande4", strlen("commande4")) == 0)
      {
            *flux = *flux + strlen("commande4");
            return 1;
      }
      else if (strncmp(*flux, "commande5", strlen("commande5")) == 0)
```

```
      {
            *flux = *flux + strlen("commande5");
            return 1;
      }
      return 0;
}
```

Alors que des structures un peu plus complexes:

```
UnIf ::= "if" "(" Commandes ")" [ UnePhrase | UnBlock ]
;
```

se traduisent ainsi :

```
int   UnIf(char **flux)
{
      char *tmp = *flux; // sauvegarde le contexte
      if (strncmp(*flux, "if", strlen("if")) == 0)//"if"
      {
            *flux = *flux + strlen("if");
            if (**flux == '(')//"("
            {
                  *flux = *flux + 1;
                  if (Commandes(flux))// appel recursif regle recursive
commande (fonction booleenne recursive sur flux)
                  {
                        if (**flux == ')')//")"
                        {
                              *flux = *flux + 1;
                              if (UnePhrase(flux) || UnBlock(flux))//
[UnePhrase | UnBlock]
                              {           return 1;}
                        }
                  }
            }
      }
      *flux = tmp;// restaure le contexte
      return 0;
}
```

Le principe reste simple:

- On remplace une définition de règle par une fonction booléenne.
- On remplace une clause par un appel à la fonction équivalente.
- Les clauses consécutives deviennent des appels de fonctions séparés par l'opérateur && en C (ou un if imbriqué).
- On remplace le OU ALORS (|) par l'opérateur || en C (ou un else if).
- On sauvegarde le contexte (le curseur dans le flux) en entrée de fonction.
- On le restaure avant chaque retour FAUX de la fonction.

En fin de récursivité on trouve les règles dites **terminales** qui lisent réellement le flux qui sont en BNF « abc », 'a' ou 'a'..'z'.