projet de compilation

Mini C++

partie 1 — à rendre le 29 novembre

version 2 — 22 octobre 2013

L'objectif de ce projet est de réaliser un compilateur pour un fragment de C++, appelé Mini C++ par la suite, produisant du code MIPS. Il s'agit d'un fragment contenant des entiers, des pointeurs et des classes, 100% compatible avec C++, au sens où tout programme de Mini C++ est aussi un programme C++ correct. Ceci permettra notamment d'utiliser un compilateur C++ existant comme référence, par exemple `g++`. Le présent sujet décrit la syntaxe de Mini C++, ainsi que la nature du travail demandé dans cette première partie.

1 Syntaxe

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle règle \rangle^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle règle \rangle^\star_t$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle règle \rangle^+$	répétition de la règle $\langle règle \rangle$ au moins une fois
$\langle règle \rangle^+_t$	répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle règle \rangle$?	utilisation optionnelle de la règle $\langle règle \rangle$ (*i.e.* 0 ou 1 fois)
($\langle règle \rangle$)	parenthésage

Attention à ne pas confondre « * » et « $^+$ » avec « `*` » et « `+` » qui sont des symboles du langage C++. De même, attention à ne pas confondre les parenthèses avec les terminaux `(` et `)`.

1.1 Conventions lexicales

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires peuvent prendre deux formes :

- débutant par `/*`, s'étendant jusqu'à `*/` et ne pouvant être imbriqués ;
- débutant par `//` et s'étendant jusqu'à la fin de la ligne.

Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

$$\begin{array}{lcl} \langle chiffre \rangle & ::= & \texttt{0–9} \\ \langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\ \langle ident \rangle & ::= & (\langle alpha \rangle \mid \texttt{\_})\,(\langle alpha \rangle \mid \langle chiffre \rangle \mid \texttt{\_})^\star \end{array}$$

Les identificateurs suivants sont des mots clés :

```
class else false for if int new NULL public return
this true virtual void while
```

Enfin les constantes littérales (entiers ou chaînes de caractères) obéissent aux expressions régulières ⟨*entier*⟩ et ⟨*chaîne*⟩ suivantes :

⟨*entier*⟩ ::= `0`
| `1–9` ⟨*chiffre*⟩*
| `0` ⟨*chiffre-octal*⟩$^+$
| `0x` ⟨*chiffre-hexa*⟩$^+$

⟨*chiffre-octal*⟩ ::= `0–7`

⟨*chiffre-hexa*⟩ ::= `0–9` | `a–f` | `A–F`

⟨*caractère*⟩ ::= tout caractère de code ASCII compris entre 32 et 127, autre que \ et "
| `\\` | `\"` | `\n` | `\t`
| `\x` ⟨*chiffre-hexa*⟩ ⟨*chiffre-hexa*⟩

⟨*chaîne*⟩ ::= `"` ⟨*caractère*⟩* `"`

1.2 Syntaxe

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal ⟨*fichier*⟩. Les associativités et précédences des divers opérateurs sont données par la table ci-dessous, de la plus faible à la plus forte précédence.

opérateur	associativité	précédence
`=`	à droite	faible
`||`	à gauche	
`&&`	à gauche	
`== !=`	à gauche	
`< <= > >=`	à gauche	↓
`+ -`	à gauche	
`* / %`	à gauche	
`!` `++` `--` `&` `*` (unaire) `+` (unaire) `-` (unaire)	à droite	
`()` `->` `.`	à gauche	forte

1.3 Indications

La grammaire des fichiers C++ est ambiguë. Pour lever les ambiguïtés, il faut notamment savoir distinguer les identificateurs qui dénotent des classes, donc des types, des autres identificateurs. Ceci ne peut être fait que dynamiquement : toute déclaration `class C ...` implique que l'identificateur `C` dénote maintenant un type [1]. Il doit y avoir une rétroaction entre l'analyseur syntaxique et l'analyseur lexical (connue sous le nom de *lexer hack*) pour que ce dernier construise des lexèmes différents pour les deux sortes d'identificateurs. Dans la grammaire qui est donnée ici, les identificateurs qui dénotent des types sont notés ⟨*tident*⟩.

1. Dans le typage, nous interdirons à une variable d'avoir le même nom qu'une classe et donc de dénoter de nouveau un identificateur usuel. C'est là une différence avec C++.

⟨*fichier*⟩ ::= (`#include <iostream>`)? ⟨*decl*⟩⋆ `EOF`
⟨*decl*⟩ ::= ⟨*decl_vars*⟩ | ⟨*decl_class*⟩ | ⟨*proto*⟩ ⟨*bloc*⟩
⟨*decl_vars*⟩ ::= ⟨*type*⟩ ⟨*var*⟩$^{+}_{,}$;
⟨*decl_class*⟩ ::= `class` ⟨*ident*⟩ ⟨*supers*⟩? `{ public :` ⟨*member*⟩⋆ `} ;`
⟨*supers*⟩ ::= `:` (`public` ⟨*tident*⟩)$^{+}_{,}$
⟨*member*⟩ ::= ⟨*decl_vars*⟩ | `virtual`? ⟨*proto*⟩ ;
⟨*proto*⟩ ::= (⟨*type*⟩ ⟨*qvar*⟩ | ⟨*tident*⟩ | ⟨*tident*⟩ `::` ⟨*tident*⟩) `(` ⟨*argument*⟩$^{\star}_{,}$ `)`
⟨*type*⟩ ::= `void` | `int` | ⟨*tident*⟩
⟨*argument*⟩ ::= ⟨*type*⟩ ⟨*var*⟩
⟨*var*⟩ ::= ⟨*ident*⟩ | `*` ⟨*var*⟩ | `&` ⟨*var*⟩
⟨*qvar*⟩ ::= ⟨*qident*⟩ | `*` ⟨*qvar*⟩ | `&` ⟨*qvar*⟩
⟨*qident*⟩ ::= ⟨*ident*⟩ | ⟨*tident*⟩ `::` ⟨*ident*⟩
⟨*expr*⟩ ::= ⟨*entier*⟩ | `this` | `false` | `true` | `NULL`
| ⟨*qident*⟩ | `*` ⟨*expr*⟩ | ⟨*expr*⟩ `.` ⟨*ident*⟩ | ⟨*expr*⟩ `->` ⟨*ident*⟩
| ⟨*expr*⟩ `=` ⟨*expr*⟩
| ⟨*expr*⟩ `(` ⟨*expr*⟩$^{\star}_{,}$ `)`
| `new` ⟨*tident*⟩ `(` ⟨*expr*⟩$^{\star}_{,}$ `)`
| `++` ⟨*expr*⟩ | `--` ⟨*expr*⟩ | ⟨*expr*⟩ `++` | ⟨*expr*⟩ `--`
| `&` ⟨*expr*⟩ | `!` ⟨*expr*⟩ | `-` ⟨*expr*⟩ | `+` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*opérateur*⟩ ⟨*expr*⟩
| `(` ⟨*expr*⟩ `)`
⟨*opérateur*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=` | `+` | `-` | `*` | `/` | `%` | `&&` | `||`
⟨*instruction*⟩ ::= `;`
| ⟨*expr*⟩ `;`
| ⟨*type*⟩ ⟨*var*⟩ (`=` ⟨*expr*⟩ | `=` ⟨*tident*⟩ `(` ⟨*expr*⟩$^{\star}_{,}$ `)`)? `;`
| `if (` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `if (` ⟨*expr*⟩ `)` ⟨*instruction*⟩ `else` ⟨*instruction*⟩
| `while (` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `for (` ⟨*expr*⟩$^{\star}_{,}$ `;` ⟨*expr*⟩? `;` ⟨*expr*⟩$^{\star}_{,}$ `)` ⟨*instruction*⟩
| ⟨*bloc*⟩
| `std::cout` (`<<` ⟨*expr_str*⟩)$^{+}$ `;`
| `return` ⟨*expr*⟩? `;`
⟨*expr_str*⟩ ::= ⟨*expr*⟩ | ⟨*chaîne*⟩
⟨*bloc*⟩ ::= `{` ⟨*instruction*⟩⋆ `}`

FIGURE 1 – Grammaire des fichiers C++.

2 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. Votre projet doit se présenter sous forme d'une archive `tar` compressée (option "`z`" de `tar`), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `minic++`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format ASCII, PostScript ou PDF.

Dans cette première partie du projet, votre compilateur `minic++` doit accepter sur sa ligne de commande l'option éventuelle `--parse-only` et exactement un fichier Mini C++, portant l'extension `.cpp`. Il doit alors réaliser l'analyse syntaxique du fichier. Si le fichier est syntaxiquement correct, votre compilateur doit terminer avec le code de sortie 0 (`exit 0`, ou terminaison normale du programme).

En cas d'erreur lexicale ou syntaxique, celle-ci doit être signalée (de la manière indiquée ci-dessous) et le compilateur doit terminer avec le code de sortie 1 (`exit 1`). En cas d'autre erreur (une erreur du compilateur lui-même), le compilateur doit terminer avec le code de sortie 2 (`exit 2`).

Lorsqu'une erreur est détectée par votre compilateur, elle doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.cpp", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez. Les localisations peuvent être obtenues pendant l'analyse syntaxique grâce aux mots-clés `$startpos` et `$endpos` de Menhir, puis conservées dans l'arbre de syntaxe abstraite.