Projet de Compilation
Licence d'informatique
—2018-2019—

Le but du projet est d'écrire un compilateur en utilisant les outils `flex` et `bison`.

Le langage source est un petit langage de programmation appelé TPC, qui ressemble à un sous-ensemble du langage C. Le langage cible est l'assembleur `nasm` 64 bits. Vous vérifierez le résultat de la compilation d'un programme en exécutant le code obtenu.

Le projet est à faire en binôme ou seul. Si vous préférez le faire en binôme mais que n'avez pas de partenaire, contactez Eric Laporte. Chaque binôme réutilisera son projet d'analyse syntaxique du premier semestre et pourra le modifier [1]. Les dates limite de rendu sont :

- le samedi 13 avril 2019 à 23h55 au plus tard pour une première version du compilateur [2] avec les fonctionnalités décrites ci-dessous dans la section Travail demandé, Rendu intermédiaire (1/4 de la note de projet) ;
- le samedi 8 juin 2019 à 23h55 au plus tard pour le compilateur complet (3/4 de la note de projet).

1 Définition informelle du langage source

Un programme TPC est une suite de fonctions. Chaque fonction est constituée de déclarations de constantes et variables (locales à la fonction), et d'une suite d'instructions. La déclaration d'une variable peut initialiser la variable avec une expression, comme dans `int i=0`. Les fonctions peuvent être récursives. Il peut y avoir des constantes et variables de portée globale. Elles sont alors déclarées avant les fonctions.

Tout programme doit comporter la fonction particulière `main` par laquelle commence l'exécution. Les types de base du langage sont `int` (entier signé codé sur 4 octets), `long` (entier signé codé sur 8 octets) et `char`. On peut convertir explicitement (*cast*) une expression dans un autre type, comme dans `char space; space=(char)32;`. Le mot clé `void` est utilisé pour indiquer qu'une fonction ne fournit pas de résultat ou n'a pas d'arguments. Les arguments d'une fonction sont transmis par valeur.

Le langage TPC utilise `print` pour afficher un entier ou un caractère. Le mot-clé `readc` permet d'obtenir un caractère (`char`) lu au clavier ; `reade` permet de lire un `int` en notation décimale.

2 Définition des éléments lexicaux

Les identificateurs sont constitués d'une lettre, suivie éventuellement de lettres, chiffres, symbole souligné ("_"). Vous pouvez fixer une longueur maximale pour un identificateur. Il y a distinction entre majuscule et minuscule. Les mots-clés comme `if`, `else`, `return`, etc., doivent être écrits en minuscules. Ils sont reconnus par l'analyseur lexical et ne peuvent pas être utilisés comme identificateurs.

Les éléments lexicaux pour les constantes numériques sont des suites de chiffres.

Les caractères littéraux dans le programme sont délimités par le symbole `'`, comme en C. Dans les caractères littéraux, la barre oblique inverse ("\") est utilisée pour déspécialiser `'` et pour spécialiser `n` et `t` : `\n` et `\t` sont le caractère fin de ligne et la tabulation.

Les commentaires sont délimités par `/*` et `*/` et ne peuvent pas être imbriqués.

Les différents opérateurs et autres éléments lexicaux sont :

`=`	: opérateur d'affectation
`+`	: addition ou plus unaire
`-`	: soustraction ou moins unaire
`*`	: multiplication
`/` et `%`	: division et reste de la division entière
`!`	: négation booléenne

(1). Si vous avez déjà validé Analyse syntaxique l'an dernier, mais pas Compilation, vous devez quand même faire le projet d'analyse syntaxique (on ne peut pas faire de compilateur sans analyseur syntaxique)

(2). Déposez votre projet sur la plateforme elearning dans la zone prévue à cet effet.

`==`, `!=`, `<`, `>`, `<=`, `>=` : les opérateurs de comparaison
`&&`, `||` : les opérateurs booléens
`;` et `,` : le point-virgule et la virgule
`(`, `)`, `{`, `}`, `[` et `]` : les parenthèses, les accolades et les crochets

Chacun de ces éléments sera identifié par l'analyse lexicale, qui devra produire une erreur pour tout élément ne faisant pas partie du lexique du langage.

3 Notations et sémantique du langage

Dans ce qui suit,
- **CARACTERE** et **NUM** désignent respectivement un caractère littéral et une constante numérique ;
- **IDENT** désigne un identificateur ;
- **TYPE** désigne un nom de type qui peut être `int` ou `char` ;
- **EQ** désigne les opérateurs d'égalité ('`==`') et d'inégalité ('`!=`') ;
- **ORDER** désigne les opérateurs de comparaison '`<`', '`<=`', '`>`' et '`>=`' ;
- **ADDSUB** désigne les opérateurs '`+`' et '`-`' (binaire ou unaire) ;
- **DIVSTAR** désigne les opérateurs '`*`', '`/`' et '`%`' ;
- **OR** et **AND** désignent les deux opérateurs booléens '`||`' et '`&&`'.
- Les mots-clés sont notés par des unités lexicales qui leur sont identiques à la casse près.

L'instruction nulle est notée '**;**'.

4 Grammaire simplifiée du langage TPC

Prog : DeclConsts DeclVars DeclFoncts ;
DeclConsts : DeclConsts **CONST** ListConst **';'**
| ;
ListConst : ListConst **',' IDENT '='** Litteral
| **IDENT '='** Litteral ;
Litteral : NombreSigne
| **CARACTERE** ;
NombreSigne : **NUM**
| **ADDSUB NUM** ;
DeclVars : DeclVars **TYPE** Declarateurs **';'**
| ;
Declarateurs : Declarateurs **','** Declarateur
| Declarateur ;
Declarateur : **IDENT**
| **IDENT '[' NUM ']'** ;
DeclFoncts : DeclFoncts DeclFonct
| DeclFonct ;
DeclFonct : EnTeteFonct Corps ;
EnTeteFonct : **TYPE IDENT '('** Parametres **')'**
| **VOID IDENT '('** Parametres **')'** ;
Parametres : **VOID**
| ListTypVar ;
ListTypVar : ListTypVar **',' TYPE IDENT**
| **TYPE IDENT** ;
Corps : '{' DeclConsts DeclVars SuiteInstr '}' ;
SuiteInstr : SuiteInstr Instr
| ;
Instr : Exp **';'**
| **';'**
| **RETURN** Exp **';'**
| **RETURN ';'**

```
            | READE  '(' IDENT ')' ';'
            | READC  '(' IDENT ')' ';'
            | PRINT '(' Exp ')' ';'
            | IF '(' Exp ')' Instr
            | IF '(' Exp ')' Instr ELSE Instr
            | WHILE  '(' Exp ')' Instr
            | '{' SuiteInstr '}'   ;
Exp         : LValue '=' Exp
            | EB   ;
EB          : EB OR TB
            | TB   ;
TB          : TB AND FB
            | FB   ;
FB          : FB EQ M
            | M   ;
M           : M ORDER E
            | E   ;
E           : E ADDSUB T
            | T    ;
T           : T DIVSTAR F
            | F    ;
F           : ADDSUB F
            | '!' F
            | '(' Exp ')'
            | LValue
            | NUM
            | CARACTERE
            | IDENT '(' Arguments ')'   ;
LValue      : IDENT
            | IDENT '[' Exp ']'   ;
Arguments   : ListExp
            |  ;
ListExp     : ListExp ',' Exp
            | Exp   ;
```

5 Sémantique

La sémantique de la plupart des expressions et instructions du langage est la sémantique habituelle en langage C.

Tout identificateur utilisé dans un programme doit être déclaré avant son utilisation et dans la partie de déclaration appropriée.

De même, la grammaire n'impose pas que le programme comporte la fonction `main`, mais l'analyse sémantique doit vérifier sa présence.

Le typage des expressions est comme en C :

- Tout `char` auquel on applique une opération (sauf `print()`) est implicitement converti en `int`.
- Toute valeur peut être interprétée comme booléenne, avec la convention que 0 représente "faux" et tout autre entier "vrai", et toute expression à sens booléen est de type `int`. En particulier, l'opérateur de négation produit comme résultat l'entier 1 quand on l'applique à l'argument 0.
- Tout `int` qu'on affecte à une LValue de type `long`, ou auquel on applique une opération dont l'autre opérande est `long`, est implicitement converti en `long`.
- Si on affecte un `int` à une LValue de type `char`, le compilateur émet un avertissement (*warning*).
- Si on affecte un `long` à une LValue de type `int` ou `char`, le compilateur émet un avertissement.

Pour le typage, les instructions `return Exp`, `reade()` et `readc()` sont considérées comme des affectations ; dans les appels de fonctions, on considère que chaque paramètre effectif correspond à une affectation.

6 Langage cible

Le langage cible est un sous-ensemble de l'assembleur `nasm` 64 bits. Les commandes autorisées sont :

`mov`, `movsx`, `call`, `ret`, `syscall`,
`add`, `sub`, `idiv`, `imul`,
`and`, `or`, `xor`,
`cmp`, `je`, `jg`, `jne`, `jng`, `jmp`,
`push`, `pop`,
`resb`, `resd`, `resq`, `db`, `dd`, `dq`.

Les registres autorisés sont :

`rax`, `rbx`,
`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`,
`rsp`, `rbp`

et leurs sous-registres `eax`, `ax`, `ah`, `al`, `ebx`, `bx`, `bh`, `bl`, `ecx`, `cx`, `ch`, `cl`, `edx`, `dx`, `dh`, `dl`, `edi`, `di`, `esi` et `si`.

7 Travail demandé

Écrire un compilateur de ce langage en utilisant `flex` pour l'analyse lexicale et `bison` pour l'analyse syntaxique et la traduction. Vous devrez modifier la grammaire simplifiée qui vous a été fournie, pour respecter la définition du langage TPC, pour lever des conflits d'analyse ou pour faciliter la traduction, mais vos modifications ne peuvent s'écarter de la définition du langage TPC que si cela l'enrichit.

Le répertoire que vous déposerez doit être organisé correctement : un répertoire pour les sources, un autre pour la documentation, un autre pour les tests... Le répertoire pour les sources doit contenir un Makefile nommé `Makefile` ou `makefile`. L'analyseur créé avec le Makefile doit être nommé `compil`. La commande suivante doit exécuter votre analyseur :

./compil [-o *prog***.asm] <** *prog***.tpc**

Le programme devra renvoyer 0 si et seulement si *prog***.tpc** est correct, c'est-à-dire ne contient aucune erreur, ni syntaxique, ni sémantique (les avertissements ne comptent pas comme des erreurs).

Déposez votre projet sur la plateforme elearning dans la zone prévue à cet effet, sous la forme d'une archive tar compressée de nom "ProjetCompilationL3_NOM1_NOM2.tar.gz", qui, au désarchivage, crée un répertoire "ProjetCompilationL3_NOM1_NOM2" contenant le projet.

Rendu intermédiaire On demande une première version du compilateur qui doit au moins :
- construire la table des symboles et y mettre au moins les variables
- détecter les variables qui sont utilisées sans être déclarées et émettre les messages d'erreur
- typer les expressions et émettre les messages d'avertissement
- redémarrer après les erreurs les plus habituelles.

On demande aussi d'écrire trois jeux de tests, un pour les programmes TPC corrects, un autre pour les programmes incorrects, et un troisième pour tester le redémarrage après erreur. Enfin, on demande un script de déploiement des tests, qui produit un rapport unique donnant les résultats de tous les tests. Pour le rendu intermédiaire, on ne demande pas de documentation.

Pour le rendu final, on demande les fonctionnalités suivantes.

Fonctions Votre compilateur doit pouvoir traiter les programmes avec plusieurs fonctions et des appels de fonctions.

Initialisation Ajouter à la grammaire des règles permettant d'initialiser les variables de type simple au moment où on les déclare, avec la syntaxe du C.

Conversion explicite Ajouter à la grammaire des règles permettant de convertir explicitement les expressions, avec la syntaxe du C.

Tableaux Il est conseillé d'écrire d'abord une première version du compilateur qui ne traite pas les tableaux, puis dans un second temps d'introduire les tableaux à une dimension, et enfin les tableaux à plusieurs dimensions. Pour pouvoir manipuler les tableaux il y a deux choses indispensables :

- Pouvoir réserver la mémoire nécessaire.
- Être capable d'accéder convenablement au contenu d'une case.

Ainsi il est nécessaire d'attacher à chaque identifiant un attribut qui spécifie la taille du tableau, et d'effectuer la réservation correspondante en début de programme.

Documentation Vous décrirez dans votre documentation vos choix et les difficultés que vous avez rencontrées.