Université Gustave Eiffel

Examen final du 26 mai 2021

Compilation

—Licence d'informatique - 3e année—

Durée : 2 h. Les quatre parties sont indépendantes. Documents papier autorisés. Appareils électroniques interdits. Le barème donné ci-dessous est indicatif.

▸ Exercice 1. *Développement en assembleur* `nasm` *64 bits (5 points)*

Écrivez en assembleur `nasm` *64 bits du code pour réaliser les tâches suivantes. Votre code peut utiliser les registres* `rax` *(sauf dans la question 1, évidemment),* `rbx`*,* `rcx` *et* `rdx` *sans sauvegarder leur contenu antérieur.*

1. *(1 point) Vérifier si les 4 octets de poids fort de* `rax` *sont à 0, sans écraser* `rax`*.*
2. *(1 point) Copier 8 octets de données depuis l'adresse contenue dans* `rsi` *vers l'adresse contenue dans* `rdi`*. Indication : l'instruction* `mov qword [rdi], qword [rsi]` *est interdite en* `nasm`*.*
3. *(3 points) Mettre en majuscules une séquence de caractères ASCII rangés à l'adresse constante* `text` *et codés sur un octet chacun. On suppose (inutile de le vérifier) que tous les caractères de la séquence représentent des lettres minuscules, sauf le dernier, qui est nul. Indication : les codes ASCII des majuscules valent 32 de moins que les codes des minuscules correspondantes.*

▸ Exercice 2. *Développement en Bison (6 points)*

La notation polonaise inverse permet de représenter des opérations arithmétiques. Le but de cet exercice et de créer une calculatrice polonaise inverse. Avec la notation habituelle, les opérations arithmétiques s'écrivent de la manière suivante : $operande_1$ $operateur$ $operande_2$, où $operateur$ est $+$, $-$, $*$ ou $/$, et où $operande_1$ et $operande_2$ sont des nombres ou des expressions. La notation polonaise inverse, elle, commence par les opérandes puis écrit l'opérateur : $operande_1$ $operande_2$ $operateur$. Par exemple, 5 + 10 s'écrit 5 10 +.

Une grammaire possible pour les expressions arithmétiques en notation polonaise inverse est :

$$
\begin{array}{lcl}
S & \rightarrow & E \\
E & \rightarrow & E\ T\ \mathit{ADDSUB} \\
 & | & T \\
T & \rightarrow & T\ F\ \mathit{DIVSTAR} \\
 & | & F \\
F & \rightarrow & \mathit{NUM}
\end{array}
$$

1. *Quel est l'avantage de la notation polonaise inverse par rapport à la notation habituelle ?*
2. *Écrivez un programme* `bison` *qui prend une expression arithmétique en notation polonaise inverse et qui affiche son résultat sur le terminal. On ne demande pas d'écrire le programme* `flex`*. Vous pouvez vous limiter aux entiers.*
3. *On permet maintenant d'utiliser des constantes nommées. La grammaire devient :*

$$
\begin{array}{lcl}
S & \rightarrow & \mathit{DeclConst}\ E \\
\mathit{DeclConst} & \rightarrow & \ldots \\
E & \rightarrow & E\ T\ \mathit{ADDSUB} \\
 & | & T \\
T & \rightarrow & T\ F\ \mathit{DIVSTAR} \\
 & | & F \\
F & \rightarrow & \mathit{NUM} \\
 & | & \mathit{Const} \\
\mathit{Const} & \rightarrow & \mathit{IDENT}
\end{array}
$$

Écrivez un programme `bison` *qui prend des déclarations de constantes éventuelles (on ne demande pas d'écrire les actions qui traduisent les déclarations) puis une expression en notation polonaise inverse, et qui produit du code en* `nasm` *64 bits qui calcule la valeur de l'expression. Les entiers seront codés sur 8 octets. Vous avez accès à une fonction* `int getOffset (char *)` *(on ne demande pas de l'écrire) qui renvoie :*

— 0 si l'identifiant passé en argument n'est pas déclaré, et

— son adresse relative par rapport à `rbp` *s'il l'est.*

L'adresse relative renvoyée est strictement négative.

Rappel : l'instruction `idiv`*, quand elle est employée avec un argument sur 8 octets, divise* `rdx:rax` *par son argument, puis place le quotient dans* `rax` *et le reste dans* `rdx`*.*

▸ Exercice 3. *Lecture de programmes Bison (4 points)*

On donne les programmes Flex et Bison suivants :

```
%{
/* mai-2021.lex */
/* Analyseur lexical lettre à lettre */
#include "mai-2021-a.h"
%}
%option nounput
%option noinput
%%
[ \t\r]+  ;
[a-zA-Z]  { yylval=yytext[0]; return ID; }
.         return yytext[0];
\n        return 0;
%%

%{
/* mai-2021-a.y */
/* listes avec Flex et Bison */
#include <stdio.h>
int yylex();
void yyerror(char *);
%}
%token ID
%%
D : L ';'       { printf(";\n"); }
L : ID          { printf("%c",$1); }
  | L ',' ID    { printf("%c",$3); }
```

```
  ;
%%
int main(int argc, char **argv) {
  yyparse();
  return 0;
}
```

1. *Étant donné la séquence* `i, j, k, l, m ;\n` *donnée en entrée, quelle sortie produisent-ils ?*
2. *Même question en remplaçant le programme Bison par celui-ci :*

```
%{
/* mai-2021-b.y */
/* listes avec Flex et Bison */
#include <stdio.h>
int yylex();
void yyerror(char *);
%}
%token ID
%%
D : L ';'          { printf(";\n"); }
L : ID             { printf("%c",$1); }
  | ID ',' L       { printf("%c",$1); }
  ;
%%
int main(int argc, char **argv) {
  yyparse();
  return 0;
}
```

3. *Même question en remplaçant le programme Bison par celui-ci :*

```
%{
/* mai-2021-c.y */
/* listes avec Flex et Bison */
#include <stdio.h>
int yylex();
void yyerror(char *);
%}
%token ID
%%
D : L ';'          { printf(";\n"); }
L : ID             { printf("%c",$1); }
  | ID { printf("%c",$1); } ',' L
  ;
%%
int main(int argc, char **argv) {
  yyparse();
  return 0;
}
```

▸ **Exercice 4. *Traduction (5 points)***

On donne la syntaxe d'un langage H, un langage de typographie pour les formules

mathématiques avec des indices et des exposants, comme LaTeX. Dans ce langage, `a_{i}` donne la formule a_i et `a_{i^{2}}^{n_{1}}` donne $a_{i^2}^{n_1}$:

$$
\begin{array}{lcl}
S & \longrightarrow & S\ F \\
S & \longrightarrow & F \\
F & \longrightarrow & G\ \text{'^'}\ \text{'\{'}\ S\ \text{'\}'} \\
F & \longrightarrow & G \\
G & \longrightarrow & \text{text}\ \text{'\_'}\ \text{'\{'}\ S\ \text{'\}'} \\
G & \longrightarrow & \text{text}
\end{array}
$$

Les lexèmes sont : text (du texte sans caractères spéciaux), `^`, `_`, `{` *et* `}`.

1. *Dessiner l'arbre de dérivation de la séquence* `a_{i^{2}}`. *Remarque : dessinez bien les liens. Dans un arbre à n nœuds, il y a $n-1$ liens père-fils.*
2. *On traduit le langage H dans un langage de bas niveau B qui permet d'afficher les formules mathématiques en plaçant les caractères à des hauteurs différentes et en leur donnant des tailles différentes. Dans le langage B, la hauteur et la taille des caractères sont indiquées dans deux registres (figure 1) :*
   - *le registre* `base` *donne la hauteur du bas des caractères par rapport au bas de la ligne, positive au-dessus de la ligne et négative au-dessous, et par défaut elle vaut 0 ;*
   - *le registre* `size` *donne la taille des caractères, dans la même unité que la hauteur, et par défaut elle vaut 100 ;*
   - *le langage B a aussi un registre de calcul,* `c`.

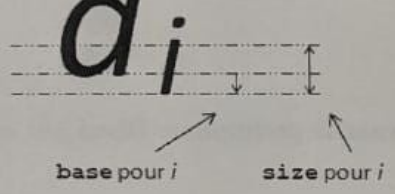

FIGURE 1 – Les registres `base` et `size` dans le langage B

Le langage B contient 6 instructions :

`display` *text*	*Afficher text à la hauteur et à la taille indiquées par* `base` *et* `size`.
`move` r_1 r_2	*Copier le registre r_2 dans le registre r_1.*
`add` r_1 r_2	*Incrémenter le registre r_1 de la valeur de r_2.*
`mult` r n	*Multiplier la valeur contenue dans le registre r par la constante n.*
`push` r	*Empiler la valeur du registre r.*
`pop` r	*Dépiler une valeur et la copier dans le registre r.*

On donne une grammaire attribuée qui traduit le langage H en langage B :

$$
\begin{array}{lcll}
S & \longrightarrow & S\ F & \\
S & \longrightarrow & F & \\
F & \longrightarrow & G\ \text{'^'}\ \text{'\{'} & \{\ \texttt{printf("push base\textbackslash n");} \\
 & & & \texttt{printf("move c, size\textbackslash n");} \\
 & & & \texttt{printf("mult c, 0.7\textbackslash n");} \\
 & & & \texttt{printf("add base, c\textbackslash n");} \\
 & & & \texttt{printf("push size\textbackslash n");}
\end{array}
$$