Projet de compilation

Le projet consiste à analyser ce langage impératif qui ressemble à Pascal, à l'interpréter, à le traduire en code à trois adresses et enfin à émuler une machine qui fait tourner ce code.

1 Langage source : définition de la syntaxe

Le langage est inspiré du langage de programmation Pascal [1]. Nous l'appellerons « *Léa* » (Langage élémentaire algorithmique).

Le langage *Léa* est défini par la grammaire suivante :

Programme

```
program:
  type_declaration_part
  variable_declaration_part
  procedure_and_function_definition_part
  TOKEN_BEGIN
  statement_list
  TOKEN_END
```

Déclaration des types nommés

```
type_declaration_part:
  'type' type_declaration_list
  | /* empty */

type_declaration_list:
  type_declaration_list type_declaration
  | type_declaration

type_declaration:
  IDENTIFIER '=' type ';'
```

Définition des types

1. http://www.fit.vutbr.cz/study/courses/APR/public/ebnf.html

```
type:
  simple_type
  | named_type
  | index_type
  | array_type
  | pointer_type

simple_type:
  'character'
  | 'integer'
  | 'boolean'

named_type:
  IDENTIFIER

index_type:
  enumerated_type
  | subrange_type

enumerated_type:
  '(' identifier_list ')'

subrange_type:
  INTEGER '..' INTEGER

array_type:
  'array' '[' range_type ']' 'of' type

range_type:
  enumerated_type
  | subrange_type
  | named_type

pointer_type:
  '^' type
```

Déclaration des variables typées

```
variable_declaration_part:
  'var' variable_declaration_list
  | /* empty */

variable_declaration_list:
  variable_declaration_list variable_declaration
  | variable_declaration

variable_declaration:
  identifier_list ':' type ';'

```

```
identifier_list:
  identifier_list ',' IDENTIFIER
  | IDENTIFIER
```

Déclaration et définition des procédures et des fonctions

```
procedure_and_function_definition_part:
  procedure_and_function_definition_list
  | /* empty */

procedure_and_function_definition_list:
  procedure_and_function_definition_list
    procedure_and_function_definition
  | procedure_and_function_definition

procedure_and_function_definition:
  procedure_and_function_definition_head
    variable_declaration_part block
  | procedure_and_function_definition_head ';'

procedure_and_function_definition_head:
  'procedure' IDENTIFIER '(' argt_part ')'
  | 'function' IDENTIFIER '(' argt_part ')' ':' type

argt_part:
  argt_list
  | /* empty */

argt_list:
  argt_list ',' argt
  | argt

argt:
  IDENTIFIER ':' type
```

Blocs

```
block:
  'begin'
  statement_list
  'end'
```

Instructions

```
statement_list:
  statement_list statement
  | statement
```

```
4
5  statement:
6    simple_statement
7    | structured_statement
8
9  simple_statement:
10   assignment_statement
11   | procedure_statement
12   | new_statement
13   | dispose_statement
14   | println_statement
15   | readln_statement
16   | return_statement
17
18 assignment_statement:
19   variable_access ':=' expression ';'
20
21 procedure_statement:
22   procedure_expression ';'
23
24 procedure_expression:
25   IDENTIFIER '(' expression_part ')'
26
27 expression_part:
28   expression_list
29   | /* empty */
30
31 expression_list:
32   expression_list ',' expression
33   | expression
34
35 new_statement:
36   'new '(' variable_access ')' ';'
37
38 dispose_statement:
39   'dispose' '(' variable_access ')' ';'
40
41 println_statement:
42   'println' '(' expression ')' ';'
43
44 readln_statement:
45   'readln' '(' expression ')' ';'
46
47 return_statement:
48   'return' '(' expression ')' ';'
49
50 structured_statement:
51   block
52   | if_statement
53   | while_statement
54
```

```
if_statement:
  'if' expression 'then' statement
  | 'if' expression 'then' statement 'else' statement

while_statement:
  'while' expression 'do' statement
```

Expressions

```
variable_access
  : IDENTIFIER
  | indexed_variable
  | variable_access '^'

indexed_variable:
  variable_access '[' expression ']'

expression:
  variable_access
  | expression '+' expression
  | expression '*' expression
  | expression '-' expression
  | expression '/' expression
  | expression '||' expression
  | expression '&&' expression
  | '-' expression
  | '!' expression
  | expression '<' expression
  | expression '<=' expression
  | expression '>' expression
  | expression '>=' expression
  | expression '=' expression
  | expression '!=' expression
  | '(' expression ')'
  | procedure_expression
  | literal
```

Expressions littérales

```
literal:
  INTEGER
  | CHARACTER
  | 'true'
  | 'false'
  | 'null'
```

Où les symboles `INTEGER`, `CHARACTER` et `IDENTIFIER` représentent les unités lexicales suivantes :

`INTEGER` Une constante numérique entière de format du langage C. Exemples : `-32768`, `32767`

`CHARACTER` Une constante représentant un caractère au format du langage C. Exemples : `'a'`, `'\''`, `'\\'`, `'\26'`, `'\x1A'`, `'\o123'`

`IDENTIFIER` Identificateur au format du langage C.

Toutes les autres unités lexicales sont représentées litéralement dans la grammaire entre apostrophes.

2 Langage source : système de type

Une expression de type est telle que
— `character`, `integer`, `boolean` sont des types
— si τ est un type, alors $pointer(\tau)$ est un type
— si τ est un type, alors $array(\tau, min, max)$ est un type, min, max des constantes entières (définissant l'intervalle du tableau)
— si τ_1 et τ_2 sont des types, alors $\tau_1 \times \tau_2$ est un type (type produit des arguments d'une fonction ou d'une procédure)
— si τ_1 et τ_2 sont des types, alors $\tau_1 \to \tau_2$ est un type (type fonction)

Bonne formation d'un code écrit en langage Léa

Un programme écrit en *Léa* est un texte engendré par cette grammaire qui vérifie les contraintes suivantes :
— Tous les identificateurs (types définis, variables, fonctions et procédures) sont déclarés avant d'être utilisés. Les fonctions et procédures peuvent être déclarées puis ensuite définies, ou seulement définies.
— Toute expression est bien typée, c'est-à-dire qu'il lui correspond une expression de type telle que définie plus haut et selon les tableaux suivants :

Opérateur binaire

	Opérande 1	Opérande 2	Valeur
$+$, $*$, $-$, $/$	`integer`	`integer`	`integer`
$<$, $>$, $<=$, $>=$	`integer` `character`	`integer` `character`	`boolean`

$=$, $!=$	`integer`
	`character`
	$pointer(\tau)$
		`integer`
		`character`
		$pointer(\tau)$
			`boolean`

$\&\&$, $\|\|$	`boolean`	`boolean`	`boolean`

Opérateur unaire

	Opérande	Valeur
$-$	`integer`	`integer`
!	`boolean`	`boolean`

Expression [2]

	Valeur
`true`, `false`	`boolean`
`null`	$pointer(\tau)$
`INTEGER`	`integer`
`CHARACTER`	`character`
`IDENTIFIER`	type de la variable
`variable_access` [$e : integer$]	type τ_1 si `variable_access` est de type $array(\tau_1, min, max)$
`IDENTIFIER` ($a_1 : \tau_1, a_2 : \tau_2, \ldots$)	type μ si le type de la fonction est $(\tau_1 \times (\tau_2 \times \ldots) \ldots) \rightarrow \mu$
`variable_access '^'`	type τ si `variable_access` est de type $pointer(\tau)$

— Toute instruction est bien typée :
  — L'affectation doit avoir des membres gauche et droit de même type.
  — Le test et la boucle ont comme argument une expression de type booléen.
  — L'instruction `return` a comme argument une expression de même type que le type de retour de la fonction qui la contient.
  — Les instruction `new` et `dispose` ont comme argument un pointeur.

2. On remarque que la bonne formation du typage statique ne suffit pas à vérifier les indices possibles des tableaux.

3 Langage source : définition de la sémantique

Idée générale

La sémantique définit l'évolution de l'état courant d'une machine abstraite sous l'effet d'une instruction (ou d'une expression qui a un effet) de *Léa*.

Un `état` Q est un tiplet (G, S, H) où G (*Global*) est un environnement statique, S (*Stack*) est une pile et H (*Heap*) un tas.

Une instruction s fait passer la machine d'un état $Q = (G, S, H)$ à un état $Q' = (G', S', H')$, ce que l'on note

$$Q/s \Rightarrow Q'$$

Une expression e fait passer la machine d'un état Q à un état Q', et produit la valeur v, ce que l'on note

$$Q/e \Rightarrow Q'/v$$

Environnements

Un environnement E est une fonction qui associe une valeur $E(x)$ à un identificateur x. Le domaine d'un environnement est l'ensemble des variables qu'il définit.

Environnement global ou *statique*

L'environnement global G a un domaine qui reste inchangé pendant toute la durée de l'exécution du programme.

Enregistrement d'activation

Un enregistrement d'activation est l'ensemble des informations relatives à l'exécution d'une procédure (resp. d'une fonction). Pour représenter un enregistrement d'activation, nous utiliserons un environnement *local* dans lequel nous distinguerons une variable *#return#* pour la valeur de retour d'une fonction, et *#save#* pour l'état de la machine avant que la procédure (resp. la fonction) soit appelée. Les arguments et variables locales à la procédure (resp. à la fonction) seront stockés dans l'environnement local.

L'enregistrement d'activation est empilé dans la pile S lors de l'appel d'une procédure ou d'une fonction, il est dépilé lorsque le contrôle retourne à l'appelant.

La valeur d'une variable x est $E(x)$ où E désigne Top si $x \in dom(Top)$ et G si $x \in dom(G) - dom(Top)$. Top désigne le sommet de la pile S et G l'environnement global.

Tas

Le tas H est une fonction partielle qui à chaque adresse $a \in \mathbb{Z}/2^{32}\mathbb{Z}$ pour laquelle $H(a)$ est définie, associe une valeur dans $\mathbb{Z}/2^{8}\mathbb{Z}$.

C'est-à-dire qu'à chaque adresse définie correspond un octet.

Le tas est modifié explicitement selon deux fonctions `new()` (allocation de mémoire dynamique) et `dispose()` (libération de la mémoire dynamique) prévues dans le langage *Léa*.

`new(x)` a comme effet d'allouer $taille(\tau)$ octets dans le tas où τ est le type de x, et d'affecter à x l'adresse du premier octet. La prochaine adresse où une nouvelle allocation est possible est augmentée de $taille(\tau)$.

`dispose(x)` a comme effet de désallouer les octets réservés à x, et d'affecter à x la valeur `null`. Aucun ramasse-miettes n'est prévu dans l'implémentation de ce projet.

La valeur pointée par une variable x dont le type est $pointeur(\tau)$ est un ensemble d'octets (dépendant du type τ) commençant à $H(E(x))$ où E est l'environnement de x.

La seule façon de connaître une adresse en langage *Léa*, est d'utiliser un pointeur p qui la désigne. Une adresse non encore allouée est égale à `null`.

Valeurs des expressions typées

Aux expressions de type correspondent un ensemble de valeurs dans $\mathbb{Z}/n\mathbb{Z}$.

Le tableau suivant donne pour chaque type, le nombre d'octets alloués en mémoire et l'encodage attendu :

Type	Image	Taille (en octets)	Encodage
`integer`	$\mathbb{Z}/2^{32}\mathbb{Z}$	2	encodés comme le sont les *signed int* en C (1 bit pour le signe, 31 pour le nombre). C'est-à-dire des relatifs compris dans l'intervalle $[-2^{32-1}, 2^{32-1}-1]$.
`character`	$\mathbb{Z}/2^{8}\mathbb{Z}$	1	ISO 8859-15
`boolean`	$\mathbb{Z}/2^{8}\mathbb{Z}$	1	false si $= 0$, true si $\neq 0$
$pointer(\tau)$	$\mathbb{Z}/2^{32}\mathbb{Z}$	4	Sans objet
$array(\tau, min, max)$	–	$k \times taille(\tau)$	Sans objet

Valeurs par défaut des expressions et instructions

Chaque expression de type τ a une valeur par défaut selon le tableau suivant

type	valeur par défaut
integer	0
character	`'\0'`
boolean	FALSE
$pointeur(\tau)$	null
$array(\tau, min, max)$	valeurs par défaut de τ sur tout le tableau

Une fonction a comme valeur pas défaut la valeur par défaut de son type de sortie. L'absence d'instruction $return(x)$ ne provoque pas d'erreur d'exécution.

Sémantique des opérateurs

La sémantique des opérateurs arithmétiques sur les entiers et les réels est une application $(\mathbb{Z}/n\mathbb{Z})^k \to \mathbb{Z}/n\mathbb{Z}$, où k est l'arité de l'opérateur.

La sémantique des opérateurs logiques sur les booléens est une application $(\mathbb{Z}/2\mathbb{Z})^k \to \mathbb{Z}/2\mathbb{Z}$, où k est l'arité de l'opérateur.

La sémantique des opérateurs de comparaison est une application $(\mathbb{Z}/n\mathbb{Z})^2 \to \mathbb{Z}/2\mathbb{Z}$.

Tous ces opérateurs ont une sémantique habituelle ; celle que l'on trouve dans le langage C. La sémantique de l'opérateur *op* sera notée $[\![op]\!]$. Exemple : $[\![2+3]\!] = [\![+]\!](2,3)$

Sémantique des expressions et instructions

La procédure `println(x)` permet de provoquer un effet vers la sortie standard. L'affichage sera alors la valeur de l'expression x selon son type, suivi du caractère `'\n'`. Les valeurs sont

- Entier relatif décimal signé pour `integer`,
- *true* ou *false* pour `boolean`,
- Le caractère `ISO 8859-15` pour `character`,
- `ARRAY` pour $array(\tau, min, max)$,
- `POINTER` pour $pointer(\tau)$.

La procédure `readln(x)` lit depuis l'entrée standard une valeur entière. x doit être de type `integer`, `character` ou `boolean` et encodé comme vu page 10.

Si p est une variables de type $pointer(\tau)$, l'expression $p\hat{}$ désigne la valeur pointée à l'adresse p. Si p a comme valeur *null*, le programme produit une erreur système.

Si t est une variables de type $array(\tau, min, max)$, l'expression $t[e]$ désigne la valeur de $t + (\sigma(e) - min)$ où $\sigma(e)$ est la valeur entière de l'expression e. Si $i < min$ ou $i > max$, la valeur est indéterminée et le programme produit une erreur système.

Les procédures et fonctions utilisent exclusivement un passage de paramètres par valeur et les valeurs ne peuvent être que de type simple[3].

Les affectations ne peuvent se faire que sur les expressions dites *accessibles*, c'est-à-dire, les variables de type simple, les expressions $t[e]$ où t est de type $array(\tau, min, max)$, les expressions $p\hat{}$ où p est de type $pointer(\tau)$.

3. Les types simples sont `integer`, `boolean` et `character`

4 Langage intermédiaire *C3A*

Le langage intermédiaire utilisé est un langage à trois adresses qui consiste en une liste d'instructions élémentaires écrites sous forme de tuples :

$$(\texttt{Étiquette}, \texttt{Opérateur}, \texttt{Opérande}_1, \texttt{Opérande}_2, \texttt{Opérande}_3)$$

Chacune des instructions a un effet sur une machine abstraite qui possède les propriétés suivantes :
- La machine a une infinité de registres r_i où sont stockées des valeurs sur $[-2^{32-1}, 2^{32-1} - 1]$.
- La machine a trois espaces de travail T, S et H. Ce sont des tableaux de 2^{32} octets, T est réservé à l'espace des données statiques, S est la pile et H le tas.
- La machine prend en entrée et produit en sortie un flux de caractères encodés en `ISO 8859-15` par l'intermédiaire de deux routines *read* et *print*.

- `Étiquette` est le numéro du tuple.
- `Opérateur` est le numéro de l'instruction (de 1 à 15).
- $\texttt{Opérande}_i$: Chaque instruction a zéro, une, deux ou trois opérandes. Un opérande est soit une étiquette, soit un nom de registre r_i, soit le nom d'un tableau T, S ou H, soit un numéral sur $[-2^{32-1}, 2^{32-1} - 1]$.

Les instructions sont les suivantes :

1. `x := y bop z`
   Instruction d'affectation où *bop* est un opérateur binaire arithmétique ($+$, $-$, $\times$, $/$), logique ($\wedge$, $\vee$), ou de comparaison ($<$, $>$, $\leqslant$, $\geqslant$, $=$, $\neq$).
2. `x := uop y`
   Instruction d'affectation où *uop* est un opérateur unaire arithmétique ($-$) ou logique ($\neg$)
3. `x := y`
   Instruction de copie
4. `goto x`
   Branchement inconditionnel ; `x` est une étiquette
5. `if x goto y`
   Branchement conditionnel. Cette instruction a pour effet de faire exécuter l'instruction étiquetée par `y` si la valeur de `x` est différente de nul, l'instruction suivante dans le cas contraire.
6. `push x`
   Empile un nouvel enregistrement d'activation de x octets dans S.
7. `pop`
   Dépile S.
8. `param x y`
   Ajoute le couple (x, y) dans l'enregistrement d'activation local

9. `call x`
   Affecte à $\#save\#$ l'adresse actuelle et lance l'exécution de la procédure (resp. de la fonction) x
10. `return x`
    Affecte la valeur x à $\#return\#$ et retourne à l'instruction $\#save\#$
11. `x := y[i]`
    Affecte à x la valeur de la mémoire située à i unités au-delà de y. y est le nom d'un tableau T, S ou H.
12. `x[i] := y`
    Affecte dans l'emplacement mémoire situé à i unités au-delà de x, la valeur y. x est le nom d'un tableau T, S ou H.
13. `x := malloc y`
    Affecte à x la valeur du contenu de l'adresse d'une zone de y octets allouée dans le tas H.
14. `free x`
    Libère la zone x allouée dans le tas H.
15. `x := *y`
    Affecte à x la valeur du contenu de l'adresse y.
16. `*x := y`
    Affecte le contenu de l'adresse x la valeur y.
17. `print x y`
    Affiche en sortie standard la valeur x. y est un nombre désignant le type de x.
    — `ISO 8859-15` sur l'octet de poids faible si y vaut 0. `Étiquette` est le numéro du tuple.
    — Un numérique sur $[-2^{32-1}, 2^{32-1} - 1]$ si y vaut 1 ;
    — `true` ou `false` selon la valeur de x si y vaut 2.
18. `x := read`
    Lit l'entrée standard pour encoder une valeur numérale sur $[-2^{32-1}, 2^{32-1} - 1]$ et l'affecte à x.

5 Travail à réaliser

1. Un analyseur lexical du langage Léa. Outre que cet analyseur doit reconnaître l'ensemble des tokens utilisés pour l'analyse syntaxique, il devra aussi reconnaitre et encoder correctement :
   — Les commentaires de type C (`/* */`)
   — Les commentaires de type C++ (`//`)
   — Les littéraux `integer` codés en décimal et en hexadécimal (*i.e.* `0x1CFE`)
   — Les littéraux `character` codés en décimal, octal, hexadécimal et écrits entre apostrophes.

   Cet analyseur lexical conservera le numéro de ligne, le numéro de colonne pour d'éventuels messages d'erreur.
2. Un analyseur syntaxique pour le langage *Léa*.

   Cette fonction teste que le texte est bien engendré par la grammaire de *Léa* et envoie un message d'erreur précis dans le cas contraire.

   Elle construit une table de symboles pour
   — Les définitions de types
   — Les variables globales
   — Les fonctions et les procédures
   — Les variables locales
3. Un analyseur sémantique pour *Léa*

   Chaque expression doit avoir un type bien formé (selon la section 2). [4]

   L'analyseur sémantique envoie un message explicite en cas de mal formation, et construit le typage des expressions dans le cas contraire.
4. Un interpréteur du langage *Léa*. Il s'agit d'une fonction qui prend en entrée un texte en *Léa*, un environnement global, un tas, une pile (G, H, S) et retourne un nouvel état (G', H', S') en produisant un effet. Les seuls effets possibles sont
   — Sortie vers la sortie erreur standard (`stderr`) d'un message d'erreur le cas échéant
   — Sortie vers la sortie standard (`stdout`) des affichages éventuels avec la procédure `println()`
5. Un compilateur de *Léa* vers *C3A*. [5]
6. Un émulateur *C3A*. Il s'agit d'une fonction qui prend en entrée un programme *C3A*, un environnement global, un tas, une pile (G, H, S) et retourne un nouvel état (G', H', S') en produisant un effet. [6]

4. La bonne formation du type dépend de la nature et du contexte de l'expression (*i.e.* comme opérande de type τ).

5. Les enseignants fourniront une bibliothèque où les structures de données pour `C3A` seront déjà réalisées.

6. Les enseignants fourniront un prototype de programme à compléter où l'analyse syntaxique du code `C3A` est déjà réalisé.

6 Modalités de réalisation

6.1 Éléments fournis par les enseignants

— Un analyseur syntaxique pour le langage `C3A` qui produit une structure de donnée de code à 3 adresses :
  — `emulator.y`
  — `emulator.l`
  — `c3a.h`
  — `c3a.c`
— Un fichier `Makefile` complet et suffisant pour l'ensemble du projet
— des fichiers d'exemple `progr-xxx.lea`, `progr-xxx.c3a` contenant des exemples de programmes en langage `Léa` ou `C3A` avec en commentaire les réalisations attendues.

Groupes Le projet doit être réalisé par des groupes de 4 étudiants [7] (ou moins à la discrétion du chargé de TD). Les groupes sont constitués sur leur proposition par l'enseignant.

Chaque groupe est représenté par l'un des trois étudiants qui est le *correspondant*. Le *correspondant* n'a aucune autorité sur le reste du groupe, son rôle est :

— De correspondre avec l'enseignant
— De correspondre avec les trois autres étudiants
— D'administrer le dépôt *GIT* ou *SVN*

6.2 Livrable

Le livrable doit être déposé par le correspondant du groupe sur un dépôt du Cremi accessible par l'enseignant selon les modalités exigées lors des séances de TD et dès la première séance de TD de la semaine du 19 mars.

La dernière version devra être déposée avant le mardi 10 avril à 23h59 et ne plus être modifié après.

Le livrable contient les fichiers suivants :

— README
  Qui contient en langue anglaise ou française les commandes nécessaires pour compiler le projet complet.
— AUTHORS
  Qui contient le noms des 4 étudiants dans l'ordre alphabétique.
— documentation.pdf
  Un rapport d'environ 10 pages qui explique le plan de votre programme, les difficultés rencontrées, les solutions apportées, les tests effectués. Ce document doit pouvoir être lu indépendamment des fichiers sources.
  Si un ou plusieurs étudiants du groupe a des difficultés à rédiger correctement en langue française parce qu'il ne serait pas francophone, il lui est autorisé de rédiger en langue anglaise une section ou un chapitre entier. Dans tous les cas, la rédaction doit être soignée, tant dans l'expression que dans la forme (erreurs typographiques, mise en page, etc.).
— Les sources

7. dans ce qui suit *étudiant*, *correspondant*, etc. valent pour *étudiant.e*, *correspondant.e* etc.

`src/Makefile`
`src/c3a.l`
`src/c3a.y`
`src/lea.l`
`src/lea.y`
`src/*.h`
`src/*.c`

Les langages suivants sont utilisés :
— `Makefile` invoqué par la commande `make` pour construire les dépendances entre fichiers.
— `Bison` qui est une implémentation GNU du compilateur de compilateur `Yacc`.
— `Flex` qui est une implémentation BSD de l'analyseur lexical `Lex`.
— Langage `C` et bibliothèques standards `ANSI-C` (`<stdio.h>` , `<stdlib.h>`, etc.)
Les commandes suivantes doivent pourvoir être lancées depuis le répertoire source :

1. `make all`
   Compile tous les exécutables ; efface les fichiers provisoires et lance tous les tests.
2. `make parser`
   Compile l'analyseur syntaxique et l'analyseur sémantique `parser`.
3. `make interpreter`
   Compile l'interpréteur du langage *Léa* `interpreter`
4. `make compiler`
   Compile le compilateur du langage `Léa compiler`
5. `make emulator`
   Compile l'émulateur du langage `C3A emulator`
6. `make <file>.err`
   Étant donné un fichier source *Léa* `<file>.lea` et `parser`, lance la commande `./parser < <file>.lea 2> <file>.err`
7. `make <file>.lea.out`
   Étant donné un fichier source *Léa* `<file>.lea` et `interpreter`, lance la commande `./interpreter < <file>.lea > <file>.lea.out`
8. `make <file>.c3a`
   Étant donné un fichier source *Léa* `<file>.lea` et `compiler`, lance la commande `./compiler < <file>.lea > <file>.c3a`
9. `make <file>.c3a.out`
   Étant donné un fichier source *Léa* `<file>.c3a` et `emulator`, lance la commande `./emulator < <file>.c3a > <file>.c3a.out`