# Projet de compilation

PREMIÈRE PARTIE

28 mars 2019

## 1 Introduction

**Le langage dont il est question dans ce projet est inspiré d'un langage classique de programmation impérative (Algol, Pascal, C, etc)** . Nous l'appellerons « *Léa* » (Langage Élémentaire Algorithmique).

Le projet consiste à analyser ce langage impératif et à le traduire en code intermédiaire.

Dans une seconde partie, qui sera donnée dans environ dix jours, nous dirons comment générer le code pour une machine Y86.

## 2 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 reconnaître 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 chaînes de caractères écrites entre `"`.

   Cet analyseur lexical conservera le numéro de ligne, le numéro de colonne pour d'éventuels messages d'erreur.
2. Compléter l'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.
3. Un analyseur sémantique pour *Léa*

   Chaque expression doit avoir un type bien formé : l'analyseur sémantique envoie un message explicite en cas de mauvaise formation d'un type, et construit le typage des expressions dans le cas contraire.

   L'analyseur construit une table de symboles pour
   — Les définitions de types
   — Les variables globales
   — Les fonctions et les procédures
   — Les variables locales
4. Un compilateur de *Léa* vers le code intermédiaire.

# 3 Éléments fournis

— Le fichier `build.xml`
— La classe principale `ubordeaux.deptinfo.compilation.project.main.Main`
— La grammaire incomplète `parser/Parser.grammar`
— Le package `ubordeaux.deptinfo.compilation.project.main.abstractTree.*` qui contient toutes les classes permettant de construire les arbres de syntaxe abstraite.
— Le package `ubordeaux.deptinfo.compilation.project.main.type.*` qui contient toutes les classes permettant de construire les types.
— Les fichiers de test `question-*/test-*.lea` qui doivent pouvoir être analysés en fonction des questions posées.

# 4 Modalités de réalisation

**Groupes** Le projet doit être réalisé par des groupes de 4 étudiants [1] et très exceptionnellement moins sur autorisation des enseignants.

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*

**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.

La dernière version devra être déposée avant le 26 avril à 23h59 et ne plus être modifiée après.

Le livrable contient les fichiers suivants :

— AUTHORS
  Qui contient le noms des 4 étudiants dans l'ordre alphabétique avec les réalisations effectuées (plusieurs réalisations par étudiant et plusieurs étudiants par réalisation sont possibles).
— documentation.pdf
  Un rapport d'environ 4 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 complètes

**Soutenances des projets** Les 29 ou 30 avril selon le calendrier fixé par Lionel Clément, votre groupe fera une démonstration commentée de 20 minutes, questions comprises.

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

# 5 Langage source : définition de la syntaxe

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

### Programme

```
program=
  type_declaration_part
  variable_declaration_part
  procedure_definition_part
  'begin'
  statement_list.stm
  'end'
```

### Déclaration des types nommés

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

type_declaration_list:
  type_declaration_list type_declaration
  | type_declaration

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

### Définition des types

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

simple_type:
  'string'
  | 'integer'
  | 'boolean'

named_type:
  IDENTIFIER

index_type:
  enumerated_type
  | subrange_type
```

```
20
21 enumerated_type:
22   '(' identifier_list ')'
23
24 subrange_type:
25   INTEGER '..' INTEGER
26   | IDENTIFIER '..' IDENTIFIER
27
28 array_type:
29   'array' '[' range_type ']' 'of' type
30
31 range_type:
32   enumerated_type
33   | subrange_type
34   | named_type
35
36 pointer_type:
37   '^' type
38
39 structure_type:
40   'struct' '{' feature_list_type '}'
41
42 feature_list_type:
43   feature_list_type feature_type
44   | feature_type
45
46 feature_type:
47   IDENTIFIER ':' type ';'
```

### Déclaration des variables typées

```
1  variable_declaration_part:
2    /* empty */
3    | 'var' variable_declaration_list
4
5  variable_declaration_list:
6    variable_declaration_list variable_declaration
7    | variable_declaration
8
9  variable_declaration:
10   identifier_list ':' type ';'
11
12 identifier_list:
13   identifier_list ',' IDENTIFIER
14   | IDENTIFIER
```

### Déclaration et définition des procédures et des fonctions

```
1  procedure_definition_part=
```

```
  | procedure_definition_list

procedure_definition_list=
  procedure_definition_list
    procedure_definition
  | procedure_definition

procedure_definition=
  procedure_definition_head block
  | procedure_definition_head TOKEN_SEMIC

procedure_definition_head=
  'procedure' IDENTIFIER '(' argt_part ')'
  | 'function' IDENTIFIER '(' argt_part.args ')' ':' type

argt_part:
  /* empty */
  | argt_list

argt_list:
  argt_list ',' argt
  | argt

argt:
  IDENTIFIER ':' type
```

### Blocs

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

### Instructions

```
statement_list:
  statement_list statement
  | statement

statement:
  simple_statement
  | structured_statement

simple_statement:
  assignment_statement
  | procedure_statement
  | new_statement
  | dispose_statement
```

```
  | println_statement
  | readln_statement
  | return_statement

assignment_statement:
  variable_access '=' expression ';'

procedure_statement:
  procedure_expression ';'

procedure_expression:
  IDENTIFIER '(' expression_part ')'

expression_part:
  /* empty */
  | expression_list

expression_list:
  expression_list ',' expression
  | expression

new_statement:
  'new' variable_access ';'

dispose_statement:
  'dispose' variable_access ';'

println_statement:
  'println' expression ';'

readln_statement:
  'readln' expression ';'

return_statement:
  'return' expression ';'

structured_statement:
  block
  | if_statement
  | while_statement
  | switch_statement

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

while_statement:
  'while' expression 'do' statement

while_statement=
  'while' expression 'do' statement
```

```
switch_statement=
  'switch' expression  'begin' case_statement_list 'end'

case_statement_list=
  case_statement_list case_statement
  | case_statement

case_statement=
  'case' IDENTIFIER ':' statement
  | 'default' ':' statement
```

### Expressions

```
variable_access:
  IDENTIFIER
  | 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
  | expression '!=' expression
  | '(' expression ')'
  | procedure_expression
  | variable_access
  | literal
```

### Expressions littérales

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

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

`IDENTIFIEUR` Identificateur au format Java.

`INTEGER` Une constante numérique entière comprise entre 0 et 0xFFFFFF.

`STRING` Une chaîne d'au plus 64 caractères entre deux `"`.

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

Les opérateurs ont les mêmes propriétés algèbriques que ce qui est défini en langage Java.

Remarque à retenir au moment d'écrire l'analyseur syntaxique : Le programme `Beaver` ne permet pas l'usage habituel de `%nonassoc`. Par conséquent, il vous est conseillé d'utiliser `%left` pour les opérateurs de comparaison.

# 6 Langage source : types

Définition : Expression de type est et n'est rien d'autre que :

— Types simples
  — « `string` », « `integer` », « `boolean` » sont des expressions de type
— Énumérés
  — une valeur d'un type énuméré est un type
  — si $\tau_1, \tau_2, \ldots, \tau_k$ sont des valeurs de type énuméré, alors « $enum(\tau_1, \tau_2, \ldots, \tau_k)$ » est un type énuméré
— Tableau
  — si $min$ et $max$ sont deux entiers avec $max > min$, « $[min..max]$ » est un type intervalle
  — si $\tau_1$ est un type, et $\tau_2$ un type intervalle ou un énuméré, alors « $array(\tau_1, \tau_2)$ » est un type tableau
— Pointeur
  — si $\tau$ est un type, alors « $pointer(\tau)$ » est un type pointeur
— Fonctions et procédures
  — « *void* » est un type
  — si $\tau_1$, $\tau_2$ ... $\tau_k$ sont des types, alors « $\tau_1 \times \tau_2 \times \ldots \tau_k$ » est un type produit
  — si $\tau_1$ est un type produit et $\tau_2$ un type, alors « $\tau_1 \rightarrow \tau_2$ » est un type fonction
  — si $\tau_1$ est un type fonction et $\tau_2$ un type produit, alors « $\tau_1(\tau_2)$ » est un type application
— Structures
  — si *name* est un nom de champ et $\tau$ est un type, alors « $feature(name, \tau)$ » est un type feature
  — si $\tau_1$, $\tau_2$ ... $\tau_k$ sont des types feature, alors « $struct(\tau_1, \tau_2 \times, \tau_k)$ » est un type structure

### Bonne formation d'un code écrit en langage Léa

Un programme écrit en *Léa* est un texte engendré par la grammaire décrite plus haut qui vérifie les contraintes suivantes :

1. Tous les identificateurs (types définis, valeurs d'énumérés, variables, fonctions et procédures) sont déclarés avant d'être utilisés. Les fonctions et procédures peuvent être déclarées puis ensuite définies. Si elles sont seulement définies, leur déclaration est implicite.
2. Toute expression est bien typée, c'est-à-dire qu'il lui correspond une expression de type telle que définie plus haut et définie selon les tableaux suivants :

**Opérateur binaire**

| | Type de opérande 1 | Type de opérande 2 | Type de la valeur |
|---|---|---|---|
| $+$, $*$, $-$, $/$ | `integer` | `integer` | `integer` |
| $<$, $>$, $<=$, $>=$ | `integer` | `integer` | `boolean` |
| $==$, $!=$ | `integer`<br>`string`<br>$pointer(\tau)$ | `integer`<br>`string`<br>$pointer(\tau)$ | `boolean` |
| &&, \|\| | `boolean` | `boolean` | `boolean` |
| [ ] | $array(\tau_1, \tau_2)$ | `integer` ou valeur d'énuméré compris dans $\tau_2$ | $\tau_1$ |
| ( ) | $\tau_1^1 \times \tau_1^2 \dots \tau_1^k \rightarrow \tau_2$ | $\tau_3^1 \times \tau_3^2 \dots \tau_3^k$ | $\tau_2$ |
| . | $struct(\tau_1 \times \tau_2 \dots \tau_k)$ | $feature(name, \tau)$ | $\tau$ |

**Opérateur unaire**

| | Type de l'Opérande | Type de la Valeur |
|---|---|---|
| _ | `integer` | `integer` |
| ! | `boolean` | `boolean` |
| ^ | $pointer(\tau)$ | $\tau$ |

**Expression**

| | Valeur |
|---|---|
| `true` | `boolean` |
| `false` | `boolean` |
| `null` | $pointer(\tau)$ |
| `INTEGER` | `integer` |
| `STRING` | `string` |
| `IDENTIFIER` | type de la variable dans son environnement courant |

3. 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.

# 7 Langage source : définition de la sémantique

### 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` | $\mathbb{Z}/2^{32}\mathbb{Z}$ | 4 | encodés comme le sont les *signed int* en C (1 bit pour le signe, 31 pour le nombre). C'est-à-dire un relatif compris dans l'intervalle $[-2^{32-1}, 2^{32-1}-1]$. |
| `string` | $\mathbb{Z}/2^{512}\ \mathbb{Z}$ | 64 | une chaîne UTF-8 sur 64 octets |
| `boolean` | $\mathbb{Z}/2^{8}\mathbb{Z}$ | 1 | 0 pour false, $\neq 0$ pour true |
| Valeur d'énuméré | $\mathbb{Z}/32\mathbb{Z}$ | 4 | encodé $2^x$ où $x$ est l'indice de la valeur comprise entre 0 et 31. Par exemple l'énuméré `(ROUGE, NOIR, BLEU, VERT, ROSE)` sera encodé $(1, 2, 4, 8, 16)$ et `VERT` sera encodé 8. |
| $pointer(\tau)$ | $\mathbb{Z}/2^{32}\mathbb{Z}$ | 4 | Encodage d'une adresse sur 4 octets. |
| $array(\tau, min, max)$ | $\mathbb{Z}/n\mathbb{Z}$ | $(max - min + 1) \times taille(\tau)$ | Encodage d'un tableau d'octets. |

### Sémantique des opérateurs

La sémantique des opérateurs arithmétiques sur les entiers est une application $(\mathbb{Z}/2^{32}\mathbb{Z})^k \to \mathbb{Z}/2^{32}\mathbb{Z}$, où $k$ est l'arité de l'opérateur .

La sémantique des opérateurs logiques sur les booléens est une application $(\mathbb{Z}/2^{8}\mathbb{Z})^k \to \mathbb{Z}/2^{8}\mathbb{Z}$, où $k$ est l'arité de l'opérateur.

La sémantique des opérateurs de comparaison est une application $(\mathbb{Z}/2^{n}\mathbb{Z})^2 \to \mathbb{Z}/2^{8}\mathbb{Z}$. Où $n$ dépend des opérandes pour l'égalité et la différence : $n = 2^{32}$ pour des entiers, $2^8$ pour des booléens et $2^{512}$ pour des chaînes de caractères.

Tous ces opérateurs ont une sémantique habituelle ; celle que l'on trouve dans le langage C.

### 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`,
  - La chaîne de caractères `UTF-8` pour `string`,
  - « 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`, `string` ou `boolean`.

- Si `p` est de type $pointer(\tau)$, l'expression `p^` désigne la valeur pointée à l'adresse `p`. Si `p` a comme valeur `null`, le programme produit un message d'erreur.

- Si `t` est de type $array(\tau, min, max)$, et `e` de type *integer* ou valeur d'énuméré, l'expression `t[e]` désigne la valeur de $t + (\sigma(e) - min)$ où $\sigma(e)$ est la valeur entière de l'expression `e`.

- Si `o` est de type $struct(\tau_1, \tau_2, \ldots \tau_k)$, et `n` un nom de champ, l'expression `o.n` désigne la valeur de $o + \omega$ où $\omega$ désigne le décalage du champ désigné par $n$.

- Les procédures et fonctions utilisent exclusivement un passage de paramètres par valeur et les valeurs ne peuvent être que de type *integer*, *boolean*, *string*, $pointer(\tau)$.

- Les affectations ne peuvent se faire que sur les expressions dites *accessibles*, c'est-à-dire, les variables de type simple, ou les expressions `t[e]` où `t` est de type $array(\tau, min, max)$, ou les expressions `p^` où `p` est de type $pointer(\tau)$.
- L'instruction `switch x begin ... end`
  es affectations ne peuvent se faire que sur les expressions dites *accessibles*, c'est-à-dire, les variables de type simple, ou les expressions `t[e]` où `t` est de type $array(\tau, min, max)$, ou les expressions `p^` où `p` est de type $pointer(\tau)$.

# 8 Langage intermédiaire

Le langage intermédiaire de notre compilateur est le langage à 3 opérateurs vu en cours et décrit *pp.* 123 et suivantes de (*Modern Compiler Implementation in Java*, *Second Edition*, Andrew W. Appel and Jens Palsberg, Cambridge University Press, 2002) donné en référence bibliographique du cours.

Nous citons le début du passage ici :

«

Here is a description of the meaning of each tree operator. First, the expressions (`Exp`), which stand for the computation of some value (possibly with side effects) :

«

— `CONST`($i$) The integer constant $i$.
— `NAME`($n$) The symbolic constant $n$ (corresponding to an assembly language label).
— `TEMP`($t$) Temporary $t$. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.
— `BINOP`($o, e_1, e_2$) The application of binary operator $o$ to operands $e_1$, $e_2$. Subexpression $e_1$ is evaluated before $e_2$. The integer arithmetic operators are `PLUS`, `MINUS`, `MUL`, `DIV`; the integer bitwise logical operators are `AND`, `OR`, `XOR`; the integer logical shift operators are `LSHIFT`, `RSHIFT`; the integer arithmetic right-shift is `ARSHIFT`. The MiniJava language has only one logical operator, but the intermediate language is meant to be independent of any source language; also, the logical operators might be used in implementing other features of MiniJava.
— `MEM`($e$) The contents of *wordSize* bytes of memory starting at address $e$ (where *wordSize* is defined in the `Frame` module). Note that when `MEM` is used as the left child of a `MOVE`, it means "store", but anywhere else it means "fetch".
— `CALL`($f, l$) A procedure call : the application of function $f$ to argument list $l$. The subexpression $f$ is evaluated before the arguments which are evaluated left to right.
— `ESEQ`($s, e$) The statement $s$ is evaluated for side effects, then $e$ is evaluated for a result.

«

The statements (stm) of the tree language perform side effects and control flow :

— `MOVE`(`TEMP` $t, e$) Evaluate $e$ and move it into temporary $t$.
— `MOVE`(`MEM`($e_1$), $e_2$) Evaluate $e_1$, yielding address $a$. Then evaluate $e_2$, and store the result into *wordSize* bytes of memory starting at $a$.
— `EXP`($e$) Evaluate $e$ and discard the result.
— `JUMP`($e, labs$) Transfer control (jump) to address $e$. The destination $e$ may be a literal label, as in `NAME`($lab$), or it may be an address calculated by any other kind of expression. For example, a C-language switch(i) statement may be implemented by doing arithmetic on $i$. The list of labels labs specifies all the possible locations that the expression e can evaluate to; this is necessary for dataflow analysis later. The common case of jumping to a known label $l$ is written as `JUMP`(`NAME`($l$), $new$ `LabelList`($l, null$)), but the `JUMP` class has an extra constructor so that this can be abbreviated as `JUMP`($l$).
— `CJUMP`($o, e_1, e_2, t, f$) Evaluate $e_1$, $e_2$ in that order, yielding values $a$, $b$. Then compare $a$, $b$ using the relational operator $o$. If the result is true, jump to $t$; otherwise jump to $f$. The relational operators are `EQ` and `NE` for integer equality and nonequality (signed or unsigned); signed integer inequalities `LT`, `GT`, `LE`, `GE`; and unsigned integer inequalities `ULT`, `ULE`, `UGT`, `UGE`.

— `SEQ`($s_1, s_2$) The statement $s_1$ followed by $s_2$.
— `LABEL`($n$) Define the constant value of name $n$ to be the current machine code address. This is like a label definition in assembly language. The value `NAME`($n$) may be the target of jumps, calls, etc.