Projet informatique 2A : Assembleur Python
—
Séance de tutorat 1

Organisation

8 semaines alternant :

- séance de tutorat (2h)
  - Explication des attendus au livrable suivant
  - Notions utiles pour la réalisation du livrable
- séance de codage (4h)
  - Aide au codage du livrable

4 livrables **dépendants l'un de l'autre successivement**.

Fonctionnement en **trinômes** :

- 1 chef·fe de projet
- 2 développeur·se·s

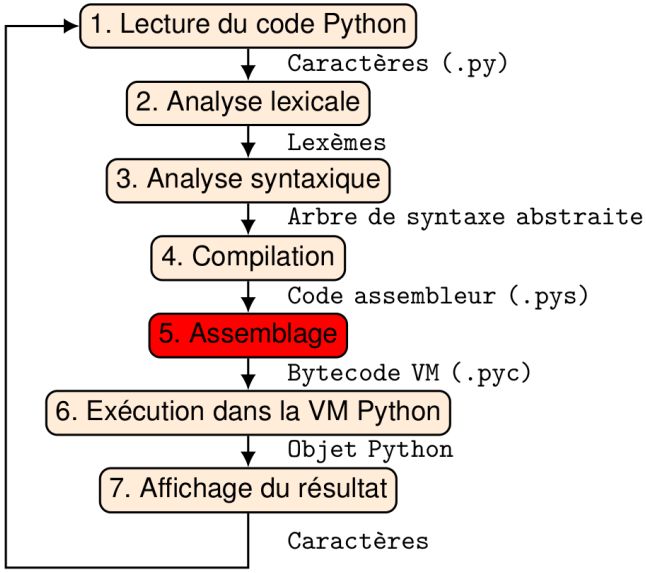
Dates de rendu : veille du prochain tutorat à minuit !

Support : Salon Riot (*cf.* page du projet pour le lien)

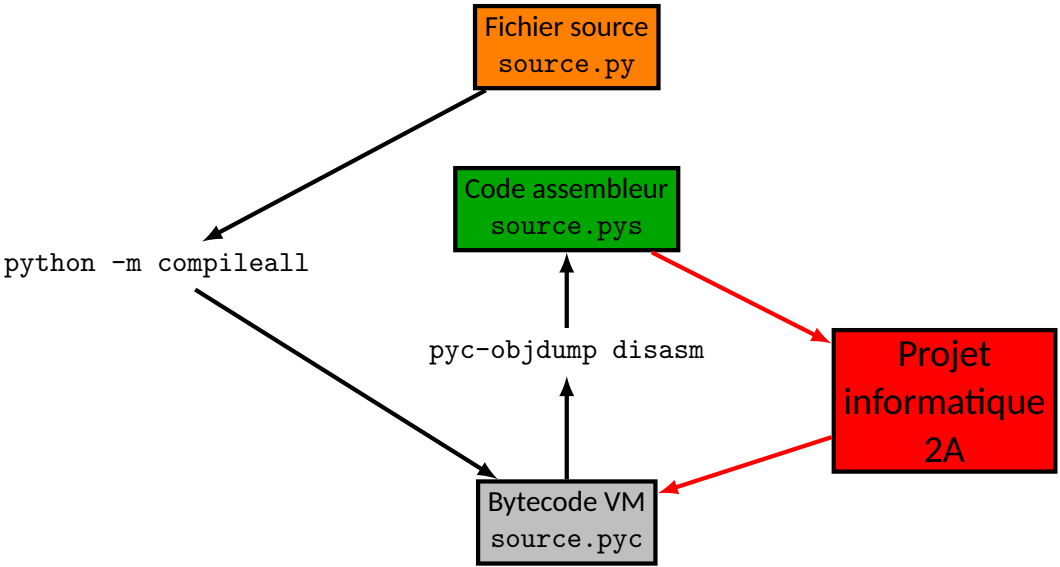
Interpréteur Python : REPL

```
$ python
Python 2.7.18 (default, Mar  8 2021, 13:02:45)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more
    information.
>>> 4+2
6
>>> exit()
$
```

Interpréteur Python : fonctionnement



Outils


Fichier source
source.py
Code assembleur
source.pys
python -m compileall
pyc-objdump disasm
Projet
informatique
2A
Bytecode VM
source.pyc

Livrables

1. Analyse lexicale (`lexer`)
   - Les **mots** du code assembleur sont-ils corrects ?
2. Analyse syntaxique (`parser`)
   - Les **phrases** formées par les **mots** du code assembleur sont-elles correctes ?
3. Génération de bytecode
   - Conversion de l'objet de code Python en bytecode.
4. Fonctions et finitions

Livrable 1 : analyse lexicale

Objectif

Objectif : reconnaître les **mots** (lexèmes) valides dans un fichier assembleur.

```
.set version_pyvm          62211
.set flags                 0x00000040
.set filename              "totor.py"
.set name                  "<module>"
.set stack_size            2

.interned
        "a"
        "b"
        "<module>"

.consts
        2
        4
        None

.names
        "a"
        "b"

.text
.line 1
        LOAD_CONST              0 # 2
        STORE_NAME              0 # "a"
```

Objectif

Objectif : reconnaître les **mots** (lexèmes) valides dans un fichier assembleur.

```
.set version_pyvm          62211
.set flags                 0x00000040
.set filename              "totor.py"
.set name                  "<module>"
.set stack_size            2

.interned
        "a"
        "b"
        "<module>"

.consts
        2
        4
        None

.names
        "a"
        "b"

.text
.line 1
        LOAD_CONST              0 # 2
        STORE_NAME              0 # "a"
```

Les types de lexèmes valides sont spécifiés dans un fichier.
Ce fichier spécifie comment reconnaître :

- les directives `.set` ou `.names`
- les nombres entiers
  - écrits en notation décimale ou hexadécimale
- les nombres réels en virgule flottante
  - potentiellement en notation exponentielle
- les chaînes de caractères
- les commentaires
- ...

Exemple de fichier de définition des lexèmes

```
# Lexems as simple regexps (first match gives lexem type!)

blank               [ \t]+
newline             \n+
comment             \#[^\n]*

colon               :
semicolon           ;

# Place keywords before identifiers!
keyword::if         if
keyword::else       else
identifier          [a-zA-Z_][a-zA-Z_0-9]*

# Numbers
# Match floats by longest regexp first!
number::float       [0-9]+\.?[0-9]*[eE][-+]?[0-9]+
number::float       [0-9]+\.[0-9]*
number::uint        [0-9]+
```

Sortie du `lexer`

```
$ cat output/file.src
# This is a comment!

if 12 :
  print a
else :
  exit

$ ./test-lexer output/lexer.conf output/file.src
 [1:0:comment] # This is a comment! [1:20:newline]

 [3:0:keyword::if] if [3:2:blank]   [3:3:number::uint] 12
     [3:5:blank]   [3:6:colon] : [3:7:newline]
 [4:0:blank]    [4:2:identifier] print [4:7:blank]   [4:8:
     identifier] a [4:9:newline]
 [5:0:keyword::else] else [5:4:blank]   [5:5:colon] : [5:6:
     newline]
 [6:0:blank]    [6:2:identifier] exit [6:6:newline]
```

Expressions régulières : à quoi bon ?

Comment reconnaitre le mot-clé `if` ?

Expressions régulières : à quoi bon ?

Comment reconnaitre le mot-clé `if` ?

- Le caractère 'i', suivi du caractère 'f'

Comment reconnaître un nombre réel en virgule flottante ?

- 123.4
- +123.4
- -123.4
- 1.23e4
- 1.23e-4
- -1.23e4
- -1.23e-4

Expressions régulières : opérateurs

- '.' (point) n'importe quel caractère,
- '\n' fin de ligne,
- '\t' tabulation,
- '\' (contre-oblique) indique de considérer le caractère qui suit (sauf 'n' et 't') comme un caractère, et non comme un opérateur,
- '?' (point d'interrogation) ce qui **précède** apparaît une fois **ou** n'apparaît pas,
- '+' (plus) ce qui **précède** apparaît au moins une fois,
- '*' (étoile) ce qui **précède** apparaît ou non,
- '^' (accent circonflexe) ce qui **suit** n'apparaît pas,

Expressions régulières : opérateurs

- '.' (point) n'importe quel caractère,
- '\n' fin de ligne,
- '\t' tabulation,
- '\' (contre-oblique) indique de considérer le caractère qui suit (sauf 'n' et 't') comme un caractère, et non comme un opérateur,
- '?' (point d'interrogation) ce qui **précède** apparaît une fois **ou** n'apparaît pas,
- '+' (plus) ce qui **précède** apparaît au moins une fois,
- '*' (étoile) ce qui **précède** apparaît ou non,
- '^' (accent circonflexe) ce qui **suit** n'apparaît pas,

Groupes de caractères : entre **crochets**, des caractères ou des **intervalles** de caractères

- '[abcde]' une des cinq premières lettres de l'alphabet,
- '[a-z]' une lettre minuscule,
- '[0-9a-fA-F]' un chiffre ou une des six premières lettres de l'alphabet en minuscule ou en majuscule.

Expressions régulières : exemples

Expression régulière	Exemples de lexème(s) valide(s)	Exemples de Lexème(s) non-valide(s)
if	“if”	“else”
else	“else”	“if”
.	’a’, ’b’, ’c’, ’\n’, etc	“AC/DC”
.*	”, “abba”, “if”, “else”, “\n\r\t:::okok”, etc	
.+	“abba”, “if”, “else”, “\n\r\t:::okok”, etc	”
0x[0-9a-fA-F]+	0x123, 0xFF	“42”, “0x”
[a-z]+	“if”, “sicom”, “sei”, “phelma”	“sei/sicom”

Fonctionnalité de base (caractères simples et opérateurs '.' et '*')

```
// regexp : l'expression reguliere
// source : la chaine a parser
int re_match( char *regexp, char *source, char **end ) {
        // Une * en deuxième position de l'expression
  if ( '*' == regexp[ 1 ] ) {
    return re_match_zero_or_more( regexp[ 0 ], regexp+2, source, end) ;
  }
// Cas général : compare la première lettre de la chaine et de l'expression
  if ( '\0' != *source &&
  ( '.' == regexp[ 0 ] || *source ==regexp[ 0 ] ) ) {
          // OK, on passe au caractère suivant de la chaine et de l'expressio
    return re_match( regexp+1, source+1, end );
  }
  // Ici, la chaine ne correspond pas a l'expression reguliere
  return 0;
}
```

Fonctionnalité de base (caractères simples et opérateurs '.' et '*')

```
// c : le caractère qui doit se trouver plusieurs fois
// regexp : l'expression reguliere  src : la chaine a parser
int re_match_zero_or_more( char c, char* rexp, char* src , char** end ) {
  char *t = src;
  // On avance dans l'analyse si on trouve c dans la chaine analysée
  while ( '\0' != *t && ( *t == c || '.' == c ) )     t++;
  // Cas d'une expression a*ab: sans cette boucle, a* consomme tous
  // les a de la chaine. Le reste de l'expression ab n'est pas parsé
  do {
    if ( re_match( rexp, t, end ) )
      return 1;
    }
  }  while( t-- > src );

  return 0;
}
```

Fonctionnalité de base : exemples

```
$ ./test-regexp-basic aa aabbbccccddddd
The start of 'aabbbccccddddd' is aa, next: 'bbbccccddddd'.

$ ./test-regexp-basic a* aabbbccccddddd
The start of 'aabbbccccddddd' is a*, next: 'bbbccccddddd'.

$ ./test-regexp-basic a*b*c* aabbbccccddddd
The start of 'aabbbccccddddd' is a*b*c*, next: 'ddddd'.

$ ./test-regexp-basic a.b* aabbbccccddddd
The start of 'aabbbccccddddd' is a.b*, next: 'ccccddddd'.

$ ./test-regexp-basic ba.b* aabbbccccddddd
The start of 'aabbbccccddddd' is *NOT* ba.b*.
```

À faire : ajout des opérateurs ’+’ et ’?’

Base de code

Le code implémentant les fonctionnalités de base (avec ’.’ et ’*’) est **fourni** dans le sujet.

Extensions du mécanisme :

- Opérateur ’+’
  - implémenter la fonction `re_match_one_or_more`
- Opérateur ’?’
  - implémenter la fonction `re_match_zero_one_or_one`

Représentation d'une expression régulière

Représentation actuelle d'une expression régulière :

- Chaîne de caractère (`char *regexp`)

Représentation d'une expression régulière

Représentation actuelle d'une expression régulière :

- Chaîne de caractère (`char *regexp`)

Problème

Impossible de gérer les groupes de caractères

À faire : structure de données pour les groupes de caractères

Définir une structure de données permettant de spécifier un groupe (ensemble) de caractères.

Par exemple :

```
struct char_group {
  char group[256];
}
```

Problème bis

Une expression régulière est spécifiée par une chaîne de caractères.

À faire : lecture d'une expression régulière

"[abcdef]" ➔ `char group[256]` contenant 'a', 'b', 'c', 'd', 'e' et 'f'.

"[a-f]" ➔ `char group[256]` contenant 'a', 'b', 'c', 'd', 'e' et 'f' (intervalle spécifié par '-').

"[a\-f]" ➔ `char group[256]` contenant 'a', '-' et 'f' (notez l'échappement).

"[a-f]+" ➔ `char group[256]` contenant 'a', 'b', 'c', 'd', 'e' et 'f' **et revenant au moins une fois**

Compléter la structure de données pour prendre en compte un éventuel **opérateur** indiquant le **nombre d'occurences**.

```
struct char_group {
  char group[256];
  /* Ajouter quelque chose ici pour indiquer
          le nombre d'occurences et la présence/absence
  */
}
```

Gestion des caractères uniques

“i” ➔ `char group[256]` contenant ’i’.

On peut maintenant représenter des expressions régulières comme :

- “i” : le caractère ’i’ une fois,
- “i+” : le caractère ’i’ une fois ou plus,
- “[a-z]” : les lettres de l’alphabet en minuscule une fois,
- “[A-Z]*” : les lettres de l’alphabet en majuscule une ou plusieurs fois ou pas du tout.

Problème bis bis

Un expression régulière est souvent formée de **plusieurs** groupes de caractères.

À faire : lecture d'une expression régulière

Exemple : "0x[0-9a-fA-F]+" (un entier en notation hexadécimale)

- `char group[256]` contenant '0'.
- `char group[256]` contenant 'x'.
- `char group[256]` contenant '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E' et 'F'.

Une expression régulière est en réalité une **liste** de `struct char_group`

Listes et files (*lists and queues*)

On vous fournit une implémentation de liste **générique** (`src/list.c`) et des exemples d'utilisation (`tests/test-list.c`).

Prenez le temps de comprendre/manipuler/tester cette liste, et d'identifier dans `list.c` :

- les fonctions de l'API, que vous utiliserez,
- les fonctions "internes", que vous n'utiliserez pas.

Une implémentation d'une **file** générique est également donnée (`src/queue.c`) mais quelques fonctions restent **à compléter**.

À faire : opérateur de négation ’^’

Exemples :

- “[a-f]” ➔ `char group[256]` contenant ’a’, ’b’, ’c’, ’d’, ’e’ et ’f’.
- “^[a-f]” ➔ `char group[256]` contenant **tous les caractères sauf** ’a’, ’b’, ’c’, ’d’, ’e’ et ’f’.
- “^a” ➔ `char group[256]` contenant **tous les caractères sauf** le ’a’.

À gérer lors de la création du groupe.

Lecture d'expressions régulières

```
list_t re_read(char *re)
```

```
$ ./test-regexp "STORE_FAST"
One in "S", one time.
One in "T", one time.
One in "O", one time.
One in "R", one time.
One in "E", one time.
One in "_", one time.
One in "F", one time.
One in "A", one time.
One in "S", one time.
One in "T", one time.

$ ./test-regexp "[\-+]?[0-9]+"
One in "-+", zero or one time.
One in "0123456789", one or more times.

$ ./test-regexp "[+\-]?[0-9]+"
One in "+-", zero or one time.
One in "0123456789", one or more times.

$ ./test-regexp "0x[0-9a-fA-F]+"
One in "0", one time.
One in "x", one time.
One in "0123456789abcdefABCDEF", one or more times.

$ ./test-regexp "^[a-f\-q-w+]+q^t*[x]?"
One not in "abcdef-qrstuvw+", one or more times.
One in "q", one time.
One not in "t", zero or more times.
One in "x", zero or one time.
```

Application

```
$ ./test-regexp "[+\-]?[0-9]+\\.?[0-9]*" "1234"
The start of '1234' is [+\-]?[0-9]+\.?[0-9]*, END: ''.

$ ./test-regexp "[+\-]?[0-9]+\\.?[0-9]*" "-+1234"
The start of '-+1234' is *NOT* [+\-]?[0-9]+\.?[0-9]*.

$ ./test-regexp "[+\-]?[0-9]+\\.?[0-9]*" "-1234.321"
The start of '-1234.321' is [+\-]?[0-9]+\.?[0-9]*, END: ''.

$ ./test-regexp "a*^bc?^d+" "accdefg"
The start of 'accdefg' is a*^bc?^d+, next: 'defg'.
```

Construction de la liste des types de lexèmes

On peut maintenant spécifier un type de lexème par une expression régulière, décrite par une chaîne de caractères.

Exemples :

- Le mot-clé `None` : `None`
- Un entier en notation hexadécimale : `0x[0-9a-fA-F]+`
- Une chaîne de caractères : `".*"`

On aimerait pouvoir les spécifier de manière **souple**, via un fichier de configuration.

À faire : lire le fichier de définition des lexèmes

```
# Lexems as simple regexps (first match gives lexem type!)

blank               [ \t]+
newline             \n+
comment             \#[^\n]*

colon               :
semicolon           ;

# Place keywords before identifiers!
keyword::if         if
keyword::else       else
identifier          [a-zA-Z_][a-zA-Z_0-9]*

# Numbers
# Match floats by longest regexp first!
number::float       [0-9]+\.?[0-9]*[eE][-+]?[0-9]+
number::float       [0-9]+\.[0-9]*
number::uint        [0-9]+
```

Fichier assembleur

```
.set version_pyvm          62211
.set flags                 0x00000040
.set filename              "totor.py"
.set name                  "<module>"
.set stack_size            2

.interned
        "a"
        "b"
        "<module>"

.consts
        2
        4
        None

.names
        "a"
        "b"

.text
.line 1
        LOAD_CONST              0 # 2
        STORE_NAME              0 # "a"
```

À faire : lecture du fichier assembleur et identification des lexèmes

Avec la **liste** des types de lexèmes construite, parcourir le fichier assembleur pour les y identifier.

Pour chaque lexème identifié, stocker :

- son type,
- sa valeur,
- le numéro de la ligne où il apparaît,
- le numéro de la colonne où il apparaît.

```
struct lexem {
  char *type;
  char *value;
  int line;
  int column;
};
```

Exemples :

- `{"number::uint", "1234", "5", "1"}`
- `{"identifier", "main", "2", "0"}`

Résultat final

```
$ cat output/file.src
# This is a comment!

if 12 :
  print a
else :
  exit

$ ./test-lexer output/lexer.conf output/file.src
 [1:0:comment] # This is a comment! [1:20:newline]

 [3:0:keyword::if] if [3:2:blank]   [3:3:number::uint] 12
    [3:5:blank]   [3:6:colon] : [3:7:newline]
 [4:0:blank]    [4:2:identifier] print [4:7:blank]   [4:8:
    identifier] a [4:9:newline]
 [5:0:keyword::else] else [5:4:blank]   [5:5:colon] : [5:6:
    newline]
 [6:0:blank]    [6:2:identifier] exit [6:6:newline]
```

A propos des listes génériques

- le type liste générique

```
                // Dans list.c, normalement non visible à l'utilisateur
struct link_t {
  void          *content;
  struct link_t *next;
};
          // Dans list.h, list_t est le type liste
typedef struct link_t * list_t;
```

- Ajout d'un élément *existant et déjà alloué* en tête de liste

```
list_t list_add_first( list_t l, void *object );
```

- Affichage de la liste

```
          // Parcours de la liste et appel de la fonction print
          // passée en paramètre à chaque maillon
int     list_print( list_t l, action_t print );
```

- Suppression du premier élément de la liste

```
list_t list_del_first( list_t l, action_t delete );
```

Les lexems

- le type lexem

```
struct lexem {
  char *type;
  char *value;
  int   line;
  int   column;
};
typedef struct lexem *lexem_t;
```

- Allocation d'un lexème à partir de valeurs

```
        // Allocation du lexem, allocation des champs type et value et co
  lexem_t lexem_new( char *type, char *value, int line, int column );
```

- Affichage d'un lexeme

```
  int     lexem_print( void *_lex );
```

- Destruction d'un lexeme

```
        // Liberation des champs type et value puis du lexem
  int     lexem_delete( void *_lex );
```

L'ajout en queue qui est bien inutile ici

```
list_t list_add_last(list_t l,void* e) {list_t p=NULL, c=NULL;
          // p est le nouveau maillon
    if ( (p=calloc( 1, sizeof (*p)))!=NULL)  { p->content=e; }
        // Si la liste est vide ou echec allocation, rien de plus a faire
    if (p==NULL || list_empty(l)) return p;
    else { c=l;
        // c s'arrete sur le dernier maillon, celui dont c->next est NULL
        while (!list_empty(c->next)) c=c->next ;
        // On accroche le nouveau à la fin
        c->next=p;
        // On retourne la liste
        return l;
    }
}
```

Utiliser plutôt *list_next()* et *list_first()* quand c'est possible

Un exemple simple

```
#include <pyas/all.h>
....
int main ( int argc, char *argv[] ) {  lexem_t unlexem;
        // Creation liste vide
  list_t l = list_new();
         // Creation d'un lexeme
  unlexem=lexem_new( "int", "42", 1, 8 );
         // Ajout de ce lexem a la liste
  l = list_add_first( l, unlexem );
       // Affichage de la liste, grace à lexem_print
  list_print(l,lexem_print));
         // Ajout en tete d'un nouveau lexeme
  l = list_add_first( l, lexem_new( "str", "ficelle", 1, 0 ) );
       // Suppression de la liste ET des maillons
  list_delete(l,lexem_delete);
```

Un exemple simple AVEC l'inclusion de tests unitaire

```
#include <pyas/all.h>
#include <unitest/unitest.h>
....
int main ( int argc, char *argv[] ) {
  list_t l = list_new();
    // initialiser le systeme de test unitaire
  unit_test( argc, argv );
    // Demarrer un test
  test_suite( "Basic test of list");

  // test une liste vide
          // boolean qui doit etre vrai   //Un commentaire
  test_assert( list_length(l) == 0, "An empty list has zero-length" );
              // Expression      // sortie ecran de l'expression   // Comment
  test_oracle( list_print( l, lexem_print ), "( )",      "Should be '( )'" );
```

Un exemple simple (suite)

```
    test_suite( "Next Basic test");
    lexem unlexem;
    unlexem = lexem_new( "int", "42", 1, 8 );
    l = list_add_first( l, unlexem );
    // test liste à un élément
    test_assert( list_length(l) == 1,"list_length() should be 1" );

    // test liste à 2 éléments
    l = list_add_last( l, lexem_new( "str", "ficelle", 1, 0 ) );
    test_assert( list_length(l) == 1,"list_length() should be 1" );

    // free memory
    list_delete( l, lexem_delete );

    exit( EXIT_SUCCESS );
}
```

Un exemple simple + tests (suite)

```
  test_suite( "Next Basic test");
  lexem unlexem;
  unlexem = lexem_new( "int", "42", 1, 8 );
  l = list_add_first( l, unlexem );
  // test liste à un élément
  test_assert( list_length(l) == 1,"list_length() should be 1" );

  // test liste à 2 éléments
  l = list_add_last( l, lexem_new( "str", "ficelle", 1, 0 ) );
  test_assert( list_length(l) == 1,"list_length() should be 1" );

  // free memory
  list_delete( l, lexem_delete );

  exit( EXIT_SUCCESS );
}
```

Compilation et execution