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

État des lieux

À l'issue du livrable 1, vous avez un *lexer* fonctionnel permettant :

- l'identification correcte des lexèmes **valides**
  - type (keyword, identifier, blank, newline, etc)
- la détection des lexèmes **invalides**
- la **localisation** des lexèmes dans le code source
  - numéro de ligne
  - numéro de colonne

Objectif du livrable 2

Développement d'un *parser* :

lexer *"Les mots du code assembleur sont-ils corrects ?"*

- ✔ Analyse **lexicale**

parser *"Les phrases formées par les mots du code assembleur sont-elles correctes ?"*

- ⏩ Analyse **syntaxique**
- ⏩ Construction d'un **objet Python**

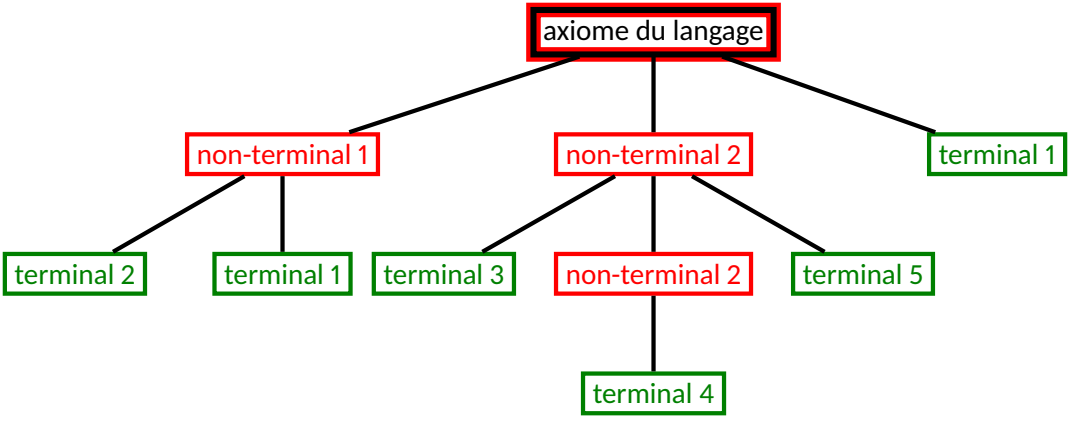
Grammaire $LL(1)$

Définition : Une grammaire $LL(n)$ a la propriété que la prochaine règle d'analyse peut être sélectionnée en utilisant au plus les n prochains lexèmes.

Dans notre cas :

- on travaille avec une grammaire $LL(1)$
- regarder **le prochain lexème** suffit pour savoir comment continuer l'analyse

Vocabulaire



Notation et opérateurs

`::=`	règle de production
`|`	alternative
`*`	répétition un nombre positif de fois
`+`	répétition un nombre strictement positif de fois
`[...]`	élément optionnel pas un groupe !
`(...)`	pour grouper des éléments
`{...}`	un lexème, obtenu de **l'analyseur lexical (livrable 1)**

- `<non-terminal>`
- `'terminal'`

Grammaire de l'assembleur Python

⟨pys⟩ ::= ⟨eol⟩ ⟨prologue⟩ ⟨code⟩

- une fin de ligne **suivie de**
- le prologue **suivi de**
- le code

⟨eol⟩ ::= ([{'blank'}] [{'comment'}] {'newline'})* [{'blank'}]

- zéro, une ou plusieurs fois :
  - un blanc (ou pas) puis
  - un commentaire (ou pas)
  - une nouvelle ligne
- puis un blanc (ou pas)

Exemple du sujet (p.81)

```
1
2  .set version_pyvm          62211
8  .interned
9          "a"
10         "b"
11         "<module>"
12
13 .consts
14         2
15         4
16         None
17
18 .names
19         "a"
20         "b"
21
22 .text
23 .line 1
24         LOAD_CONST              0 # 2
25         STORE_NAME              0 # "a"
29 .line 4
30         LOAD_NAME               0 # "a"
31         LOAD_NAME               1 # "b"
32         COMPARE_OP              0 # "<"
33         POP_JUMP_IF_FALSE       label_0
39 .line 7
40 label_0:
41         LOAD_NAME               1 # "b"
42         PRINT_ITEM
43         PRINT_NEWLINE
44 label_1:
45 coucou :
46         LOAD_CONST              2 # None
47         RETURN_VALUE
```

⟨*pys*⟩ ::= ⟨*eol*⟩ ⟨*prologue*⟩ ⟨*code*⟩

⟨*prologue*⟩ ::= ⟨*set-directives*⟩ ⟨*interned-strings*⟩ ⟨*constants*⟩ ⟨*names*⟩ [⟨*varnames*⟩ ⟨*freevars*⟩ ⟨*cellvars*⟩]

⟨*set-version-pyvm*⟩ ::= {'dir::set'} {'blank'} {'version_pyvm'} {'blank'} {'integer::dec'} ⟨*eol*⟩

⟨*interned-strings*⟩ ::= {'dir::interned'} ⟨*eol*⟩ ({'string'} ⟨*eol*⟩)*

⟨*constants*⟩ ::= {'dir::consts'} ⟨*eol*⟩ (⟨*constant*⟩ ⟨*eol*⟩)*

⟨*constant*⟩ ::= {'integer'} | {'float'} | {'string'} | {'pycst'} | ⟨*list*⟩ | ⟨*tuple*⟩

⟨*list*⟩ ::= {'brack::left'} ⟨*constant*⟩* {'brack::right'}

⟨*tuple*⟩ ::= {'paren::left'} ⟨*constant*⟩* {'paren::right'}

⟨*names*⟩ ::= {'dir::names'} ⟨*eol*⟩ ({'string'} ⟨*eol*⟩)*

⟨*code*⟩ ::= {'dir::text'} ⟨*eol*⟩ (⟨*assembly-line*⟩ ⟨*eol*⟩)*

⟨*assembly-line*⟩ ::= ⟨*insn*⟩
| ⟨*source-lineno*⟩
| ⟨*label*⟩

⟨*label*⟩ ::= {'symbol'} {'blank'} {'colon'}

⟨*source-lineno*⟩ ::= {'dir::line'} {'blank'} {'integer::dec'}

⟨*insn*⟩ ::= {'insn::0'}
| {'insn::1'} ({'integer::dec'} | {'symbol'})

⟨*eol*⟩ ::= ([{'blank'}] [{'comment'}] {'newline'})* [{'blank'}]

Descente récursive

En partant de l'axiome du langage, on **descend** dans les non-terminaux, jusqu'aux terminaux.

On écrit **une fonction pour chaque non-terminal**, chargée de le reconnaître :

- Non-terminal **reconnu** : retourne **0**
- Non-terminal **non reconnu** : retourne **-1**

Exemple 1 : l'axiome du langage des expressions arithmétiques

⟨arith-expr⟩ ::= ⟨term⟩ (({'op::sum::plus'} | {'op::sum::minus'})⟨term⟩)*

```
int parse_arith_expr( list_t *lexems ) {
  printf( "Parsing arithmetic expression\n" );

  if ( -1 == parse_term( lexems ) ) {
    return -1;
  }

  while ( next_lexem_is( lexems, "op::sum" ) ) {
    lexem_advance( lexems );

    if ( -1 == parse_term( lexems ) ) {
      return -1;
    }
  }

  return 0;
}
```

Attention !

`parse_term` est la fonction associée au non-terminal ⟨term⟩, qui désigne **un terme** de l'expression arithmétique, pas un terminal au sens de l'analyse lexicale.

Exemple 2 : le non-terminal ⟨factor⟩

⟨factor⟩ ::= {'number'} | {'identifier'} | {'paren::left'} ⟨arith-expr⟩ {'paren::right'}

```
1  int parse_factor( list_t *lexems ) {
2    printf( "Parsing factor\n" );
3
4    if ( next_lexem_is( lexems, "number" ) ||
5         next_lexem_is( lexems, "identifier" ) ) {
6      printf( "* Found factor:" );
7      lexem_print( lexem_peek( lexems ) );
8      printf( "\n" );
9      lexem_advance( lexems );
10     return 0;
11   }
12
13   if ( next_lexem_is( lexems, "paren::left" ) ) {
14     lexem_advance( lexems );
15     if ( -1 == parse_arith_expr( lexems ) ) {
16       return -1;
17     }
18
19     if ( next_lexem_is( lexems, "paren::right" ) ) {
20       lexem_advance( lexems );
21       return 0;
22     }
23
24     print_parse_error( "Missing right parenthesis", lexems );
25     return -1;
26   }
27
28   print_parse_error( "Unexpected input", lexems );
29   return -1;
30 }
```

Correspondance entre EBNF et C

EBNF	C
Concaténation	Suite d'instructions
Alternative `'|'`	`if (... || ...) {...; return...;}`
Répétition zéro, une ou plus. fois `'*'`	`while (...) {...}`
Répétition une fois au moins `'+'`	`if (!... {return -1;})` *puis* `while (...){...}`
Option `'[...]'`	`if ( ... {...} )`
Non-terminal	Appel de la fonction associée

Fonctions utiles

Comme indiqué p.42, ces fonctions vous seront particulièrement utiles :

- `lexem_t lexem_peek(list_t *lexems);`
  - Renvoyer le prochain lexème **sans l'enlever de la liste**
- `lexem_t lexem_advance(list_t *lexems);`
  - Renvoyer le prochain lexème **en l'enlevant de la liste**
- `int next_lexem_is(list_t *lexems, char *type);`
  - détermine si le prochain lexème utile est bien du type demandé
  - voir les fonctions `lexem_type_strict` et `lexem_type` (p.36)
- `void print_parse_error(char *msg, list_t *lexems);`
  - afficher le message d'erreur `msg` et les coordonnées du lexème en tête de `*lexems`.

Fonctions utiles

Comme indiqué p.42, ces fonctions vous seront particulièrement utiles :

- `lexem_t lexem_peek(list_t *lexems);`
  - Renvoyer le prochain lexème **sans l'enlever de la liste**
- `lexem_t lexem_advance(list_t *lexems);`
  - Renvoyer le prochain lexème **en l'enlevant de la liste**
- `int next_lexem_is(list_t *lexems, char *type);`
  - détermine si le prochain lexème utile est bien du type demandé
  - voir les fonctions `lexem_type_strict` et `lexem_type` (p.36)
- `void print_parse_error(char *msg, list_t *lexems);`
  - afficher le message d'erreur `msg` et les coordonnées du lexème en tête de `*lexems`.

Certaines de ces fonctions feraient bien d'ignorer les **lexèmes inutiles** : blancs et commentaires.

Ces fonctions font avancer le pointeur de tête de la liste, mais ne libèrent pas la **mémoire**.
Cela sera fait une fois pour toutes à la fin du `main`.

Où en sommes-nous ?

Avec tout ça, on peut **vérifier la syntaxe** (que les **phrases** du code source sont **correctes**).

On va maintenant construire un **objet** représentant ce qu'on a lu et qui soit manipulable en C.
Il constituera la **sortie** de l'analyseur syntaxique (*parser*) : **un objet (de code) Python**.

Structure de données pour un objet Python

```
typedef unsigned int pyobj_type;

struct pyobj_t;
typedef struct pyobj *pyobj_t;


struct pyobj {

  pyobj_type      type;
  unsigned int    refcount;

  union {
    struct {
      pyobj_t        *value;
      int32_t         size;
    }               list;

    struct {
      char           *buffer;
      int             length;
    }              string;

    py_codeblock   *codeblock;

    union {
      int32_t         integer;
      int64_t         integer64;
      double          real;
      struct {
        double          real;
        double          imag;
      }               complex;
    }              number;
  }               py;
};
```

Structure de données pour un objet Python

```
typedef unsigned int pyobj_type;

struct pyobj_t;
typedef struct pyobj *pyobj_t;


struct pyobj {

  pyobj_type      type;
  unsigned int    refcount;

  union {
    struct {
      pyobj_t        *value;
      int32_t         size;
    }               list;

    struct {
      char           *buffer;
      int             length;
    }               string;

    py_codeblock   *codeblock;

    union {
      int32_t         integer;
      int64_t         integer64;
      double          real;
      struct {
        double          real;
        double          imag;
      }               complex;
    }               number;
  }               py;
};
```

Structure de données pour un objet Python

```
typedef unsigned int pyobj_type;

struct pyobj_t;
typedef struct pyobj *pyobj_t;


struct pyobj {

  pyobj_type      type;
  unsigned int    refcount;

  union {
    struct {
      pyobj_t         *value;
      int32_t          size;
    }                list;

    struct {
      char            *buffer;
      int              length;
    }                string;

    py_codeblock    *codeblock;

    union {
      int32_t          integer;
      int64_t          integer64;
      double           real;
      struct {
        double           real;
        double           imag;
      }                complex;
    }                number;
  }                py;
};
```

Structure de données pour un objet Python

```
1  typedef unsigned int pyobj_type;
2
3  struct pyobj_t;
4  typedef struct pyobj *pyobj_t;
5
6
7  struct pyobj {
8
9    pyobj_type      type;
10   unsigned int    refcount;
11
12   union {
13     struct {
14       pyobj_t        *value;
15       int32_t         size;
16     }               list;
17
18     struct {
19       char           *buffer;
20       int             length;
21     }               string;
22
23     py_codeblock   *codeblock;
24
25     union {
26       int32_t         integer;
27       int64_t         integer64;
28       double          real;
29       struct {
30         double          real;
31         double          imag;
32       }               complex;
33     }               number;
34   }                 py;
35 };
```

Un objet Python est représenté par une **union étiquetée :**

- une étiquette
  - `type` (l.9)
- une `union` (l.12)
  - `list` **OU**
  - `string` **OU**
  - `py_codeblock` **OU**
  - `number`.

Principe de fonctionnement

La **valeur** de l'étiquette indique **quel champ** de l'union est utilisé.

Voir Table C.1 (p.146) pour les valeurs possibles de l'étiquette.

```
.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"
.line 2
        LOAD_CONST              1 # 4
        STORE_NAME              1 # "b"
.line 4
        LOAD_NAME               0 # "a"
        LOAD_NAME               1 # "b"
        COMPARE_OP              0 # "<"
        POP_JUMP_IF_FALSE       label_0
.line 5
        LOAD_NAME               0 # "a"
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 7
label_0:
        LOAD_NAME               1 # "b"
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
coucou :
        LOAD_CONST              2 # None
        RETURN_VALUE
```

```
typedef struct {
  int          version_pyvm;
  struct {
    uint32_t    arg_count;
    uint32_t    local_count;
    uint32_t    stack_size;
    uint32_t    flags;
  }            header;
  pyobj_t      parent;
  struct {
    struct {
      uint32_t   magic;
      time_t     timestamp;
      uint32_t   source_size;
    }           header;
    struct {
      pyobj_t     interned;
      pyobj_t     bytecode;
      pyobj_t     consts;
      pyobj_t     names;
      pyobj_t     varnames;
      pyobj_t     freevars;
      pyobj_t     cellvars;
    }           content;
    struct {
      pyobj_t    filename;
      pyobj_t    name;
      uint32_t   firstlineno;
      pyobj_t    lnotab;
    }           trailer;
  }            binary;
} py_codeblock;
```

```
.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"
.line 2
        LOAD_CONST              1 # 4
        STORE_NAME              1 # "b"
.line 4
        LOAD_NAME               0 # "a"
        LOAD_NAME               1 # "b"
        COMPARE_OP              0 # "<"
        POP_JUMP_IF_FALSE       label_0
.line 5
        LOAD_NAME               0 # "a"
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 7
label_0:
        LOAD_NAME               1 # "b"
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
coucou :
        LOAD_CONST              2 # None
        RETURN_VALUE
```

```
typedef struct {
  int          version_pyvm;
  struct {
    uint32_t    arg_count;
    uint32_t    local_count;
    uint32_t    stack_size;
    uint32_t    flags;
  }            header;
  pyobj_t      parent;
  struct {
    struct {
      uint32_t   magic;
      time_t     timestamp;
      uint32_t   source_size;
    }           header;
    struct {
      pyobj_t     interned;
      pyobj_t     bytecode;
      pyobj_t     consts;
      pyobj_t     names;
      pyobj_t     varnames;
      pyobj_t     freevars;
      pyobj_t     cellvars;
    }           content;
    struct {
      pyobj_t    filename;
      pyobj_t    name;
      uint32_t   firstlineno;
      pyobj_t    lnotab;
    }           trailer;
  }            binary;
} py_codeblock;
```

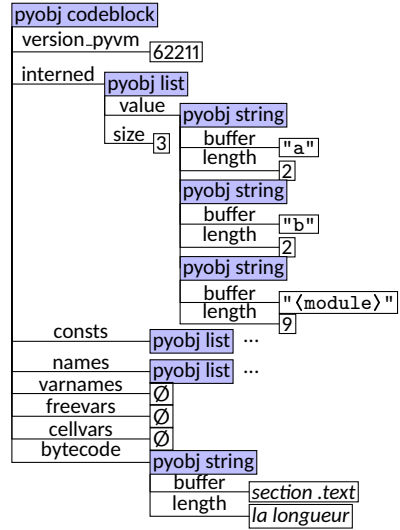
```
.set version_pyvm          62211
.set flags                 0x00000040
.set filename              "totor.py"
.set name                  "<module>"
.set stack_size            2

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

.consts
        2
        4
        None

.names
        "a"
        "b"

.text
.line 1
        LOAD_CONST              0 # 2
        STORE_NAME              0 # "a"
.line 2
        LOAD_CONST              1 # 4
        STORE_NAME              1 # "b"
.line 4
        LOAD_NAME               0 # "a"
        LOAD_NAME               1 # "b"
        COMPARE_OP              0 # "<"
        POP_JUMP_IF_FALSE       label_0
.line 5
        LOAD_NAME               0 # "a"
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 7
label_0:
        LOAD_NAME               1 # "b"
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
coucou :
        LOAD_CONST              2 # None
        RETURN_VALUE
```

```
typedef struct {
  int          version_pyvm;
  struct {
    uint32_t     arg_count;
    uint32_t     local_count;
    uint32_t     stack_size;
    uint32_t     flags;
  }            header;
  pyobj_t      parent;
  struct {
    struct {
      uint32_t   magic;
      time_t     timestamp;
      uint32_t   source_size;
    }            header;
    struct {
      pyobj_t      interned;
      pyobj_t      bytecode;
      pyobj_t      consts;
      pyobj_t      names;
      pyobj_t      varnames;
      pyobj_t      freevars;
      pyobj_t      cellvars;
    }            content;
    struct {
      pyobj_t    filename;
      pyobj_t    name;
      uint32_t   firstlineno;
      pyobj_t    lnotab;
    }            trailer;
  }            binary;
} py_codeblock;
```

Se concentrer pour l'instant sur `binary.content`

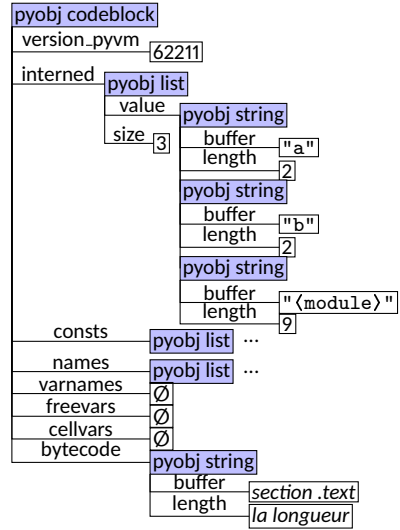
```
.set version_pyvm          62211
.set flags                 0x00000040
.set filename              "totor.py"
.set name                  "<module>"
.set stack_size            2

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

.consts
        2
        4
        None

.names
        "a"
        "b"

.text
.line 1
        LOAD_CONST              0 # 2
        STORE_NAME              0 # "a"
.line 2
        LOAD_CONST              1 # 4
        STORE_NAME              1 # "b"
.line 4
        LOAD_NAME               0 # "a"
        LOAD_NAME               1 # "b"
        COMPARE_OP              0 # "<"
        POP_JUMP_IF_FALSE       label_0
.line 5
        LOAD_NAME               0 # "a"
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 7
label_0:
        LOAD_NAME               1 # "b"
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
coucou :
        LOAD_CONST              2 # None
        RETURN_VALUE
```

```
typedef struct {
  int           version_pyvm;
  struct {
    uint32_t    arg_count;
    uint32_t    local_count;
    uint32_t    stack_size;
    uint32_t    flags;
  }             header;
  pyobj_t       parent;
  struct {
    struct {
      uint32_t    magic;
      time_t      timestamp;
      uint32_t    source_size;
    }           header;
    struct {
      pyobj_t     interned;
      pyobj_t     bytecode;
      pyobj_t     consts;
      pyobj_t     names;
      pyobj_t     varnames;
      pyobj_t     freevars;
      pyobj_t     cellvars;
    }           content;
    struct {
      pyobj_t     filename;
      pyobj_t     name;
      uint32_t    firstlineno;
      pyobj_t     lnotab;
    }           trailer;
  }             binary;
} py_codeblock;
```



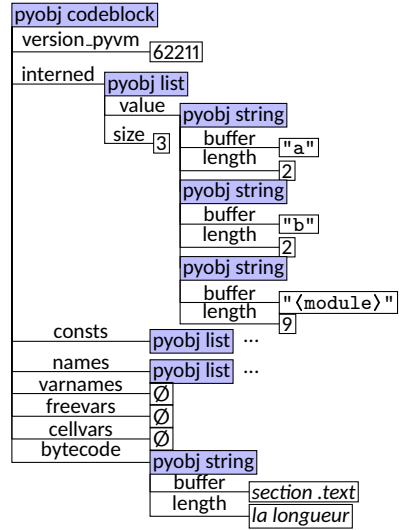
```
.set version_pyvm          62211
.set flags                 0x00000040
.set filename              "totor.py"
.set name                  "<module>"
.set stack_size            2

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

.consts
        2
        4
        None

.names
        "a"
        "b"

.text
.line 1
        LOAD_CONST              0 # 2
        STORE_NAME              0 # "a"
.line 2
        LOAD_CONST              1 # 4
        STORE_NAME              1 # "b"
.line 4
        LOAD_NAME               0 # "a"
        LOAD_NAME               1 # "b"
        COMPARE_OP              0 # "<"
        POP_JUMP_IF_FALSE       label_0
.line 5
        LOAD_NAME               0 # "a"
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 7
label_0:
        LOAD_NAME               1 # "b"
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
coucou :
        LOAD_CONST              2 # None
        RETURN_VALUE
```

```
typedef unsigned int pyobj_type;

struct pyobj_t;
typedef struct pyobj *pyobj_t;


struct pyobj {

  pyobj_type       type;
  unsigned int     refcount;

  union {
    struct {
      pyobj_t         *value;
      int32_t          size;
    }                 list;

    struct {
      char            *buffer;
      int              length;
    }                 string;

    py_codeblock    *codeblock;

    union {
      int32_t          integer;
      int64_t          integer64;
      double           real;
      struct {
        double           real;
        double           imag;
      }                complex;
    }                 number;
  }                 py;
};
```



```
.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"
.line 2
        LOAD_CONST              1 # 4
        STORE_NAME              1 # "b"
.line 4
        LOAD_NAME               0 # "a"
        LOAD_NAME               1 # "b"
        COMPARE_OP              0 # "<"
        POP_JUMP_IF_FALSE       label_0
.line 5
        LOAD_NAME               0 # "a"
        PRINT_ITEM
        PRINT_NEWLINE
        JUMP_FORWARD            label_1
.line 7
label_0:
        LOAD_NAME               1 # "b"
        PRINT_ITEM
        PRINT_NEWLINE
label_1:
coucou :
        LOAD_CONST              2 # None
        RETURN_VALUE
```

```
typedef struct {
  int           version_pyvm;
  struct {
    uint32_t    arg_count;
    uint32_t    local_count;
    uint32_t    stack_size;
    uint32_t    flags;
  }             header;
  pyobj_t       parent;
  struct {
    struct {
      uint32_t    magic;
      time_t      timestamp;
      uint32_t    source_size;
    }           header;
    struct {
      pyobj_t     interned;
      pyobj_t     bytecode;
      pyobj_t     consts;
      pyobj_t     names;
      pyobj_t     varnames;
      pyobj_t     freevars;
      pyobj_t     cellvars;
    }           content;
    struct {
      pyobj_t     filename;
      pyobj_t     name;
      uint32_t    firstlineno;
      pyobj_t     lnotab;
    }           trailer;
  }             binary;
} py_codeblock;
```



Sujet p.89: *"L'objet de code Python retourné ne contiendra pas encore le bytecode ni lnotab. À la place, vous stockerez* ***la liste des lexèmes correspondant à la section .text***"

Construction d'un objet `pyobj` de type `list`

```
8  .interned
9          "a"
10         "b"
11         "<module>"
```

```
struct {
  pyobj_t          *value;
  int32_t           size;
}                  list;
```

⟨interned-strings⟩ ::= {'dir::interned'} ⟨eol⟩ ({'string'}⟨eol⟩)*

Pour construire la liste :

- parser les chaînes de caractères dans une liste générique temporaire **en conservant l'ordre**,
- obtenir la **taille** de cette liste (avec la fonction `list_length`),
- allouer le tableau (champ `value`) et fixer la taille (champ `size`),
- copier les éléments de la liste temporaire dans le tableau.

Et ensuite ?

Au livrable 3 vous aurez à :

- **générer le bytecode**, interprétable par la VM Python :
  - en travaillant sur la section `.text`, qui est pour l'instant une longue liste de lexèmes,
  - (pensez à garder la liste des lexèmes associée, construite au livrable 1).

- **sérialiser** l'objet Python :
  - en plaçant les bonnes informations au bon endroit (en-tête, etc)
  - en générant les instructions de la VM Python à partir de l'objet `bytecode`
  - en écrivant le résultat dans un fichier `.pyc`

Tests d'intégration ? - sujet Annexe A. page 105

- A l'incrément 1, vous avez (normalement) écrit des *tests unitaires* dans le répertoire `tests/unit`, au moyen des fonctions du module `unitest.h / unitest.c` fourni.
- Pour finaliser cet incrément, il convient de réaliser maintenant des *tests d'intégration*.
- Il s'agit de tester non plus une ou plusieurs fonctions C au moyen de code C, mais l'intégration de fonctionnalités dans un programme exécutable, en exécutant ce programme plusieurs fois, sur plusieurs données d'entrée (qu'on appelle *jeu de tests*).
- Pour cela, le bootstrap fourni dans `tests/integration` :
  - un script Python, qui automatise l'exécution d'un de vos programme sur tous les tests d'un jeu de tests et, pour chaque test, la comparaison des résultats de votre programme (ses sorties dans le Terminal et son code de retour) avec les résultats attendus pour le test.
  - un ensemble de jeux de tests, pour chacun des programmes de l'incrément 1 :
    - votre programme qui réalise le *parsing* d'une expression régulière passée en argument dans le terminal ;
    - votre programme qui réalise le *parsing* d'une expression régulière et d'une chaîne de caractères passée en argument dans le terminal ;
    - votre programme *lexer* qui affiche la liste des lexèmes d'un code assembleur Python.

Tests d'intégration ? - sujet Annexe A. page 105

- Comment faire en pratique ?
  - Lire la documentation du script de test Python3 fourni : `bootstrap/tests/integration/README.pdf`
  - Exemple : vérifier que le *matching* d'expressions régulières basiques (telles que fournies dans le bootstrap : pas de groupes de caractères, opérateurs `.` et `*` uniquement) fonctionnent toujours :
    - Aller dans le répertoire `bootstrap/tests/integration`
    - Exécuter le script de test Python3 fourni, en lui passant votre programme `regexp-match` et en lui indiquant le jeu de tests concernés, c'est à dire le répertoire `00_test_regexp_basic` :
      ```
      python3 ./execute_tests.py runtest ../../bin/regexp-match.exe
      ./00_test_regexp_basic/
      ```
    - Le script vous indique quel test a fonctionné et quel test est en erreur.
    - Pour analyser les résultats plus finement, comparer les sorties attendues aux sorties obtenues avec l'outil `meld` :
      ```
      meld ./00_test_regexp_basic/ ./00_test_regexp_basic_result
      ```

Tests d'intégration ? - sujet Annexe A. page 105

```
######################################################################
TEST RESULTS SUMMARY
######################################################################
Tests successful :
  ../../bin/regexp-match.exe  'aa' 'aabbbccccddddd'
  ../../bin/regexp-match.exe  'a*' 'aabbbccccddddd'
  ../../bin/regexp-match.exe  'a*b*c*' 'aabbbccccddddd'
  ../../bin/regexp-match.exe  'a.b*' 'aabbbccccddddd'
  ../../bin/regexp-match.exe  'ba.b*' 'aabbbccccddddd'
  ../../bin/regexp-match.exe  'ba.b*' 'bac OK'
  ../../bin/regexp-match.exe  'STORE_FAST' 'STOerror'
  ../../bin/regexp-match.exe  'STORE_FAST' 'STORE_FAST___OK'
######################################################################
*** Number of tests successful  : 8 out of 8
*** Number of tests with errors : 0 out of 8
***  Among which :
***   Number of tests with segfault, signal, or user-terminated : 0
***   Number of tests with incorrect return code                : 0
***   Number of tests with incorrect return output              : 0
######################################################################
You may want to compare test oracles and test results by running :
  meld 00_test_regexp_basic 00_test_regexp_basic_result
```