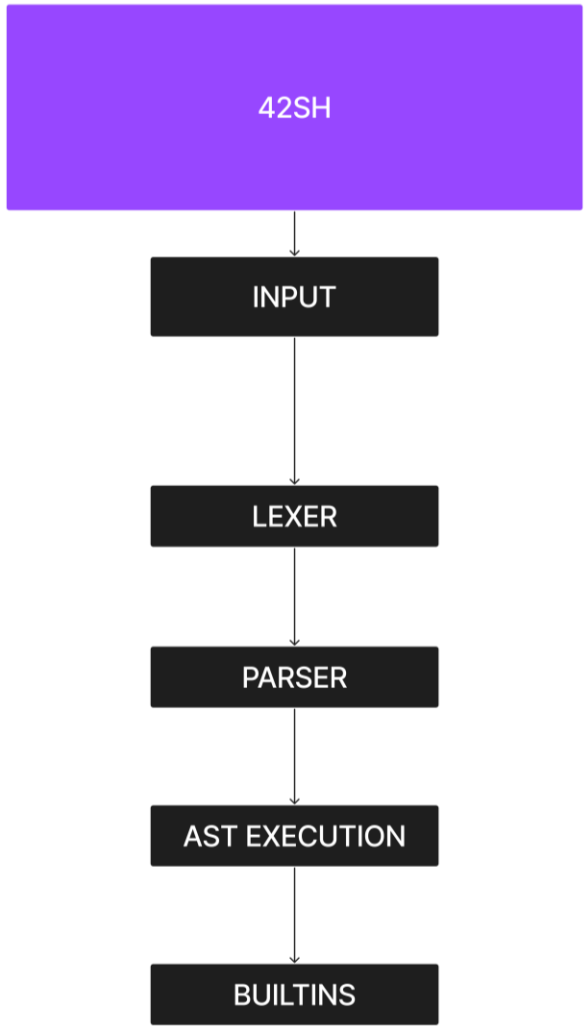
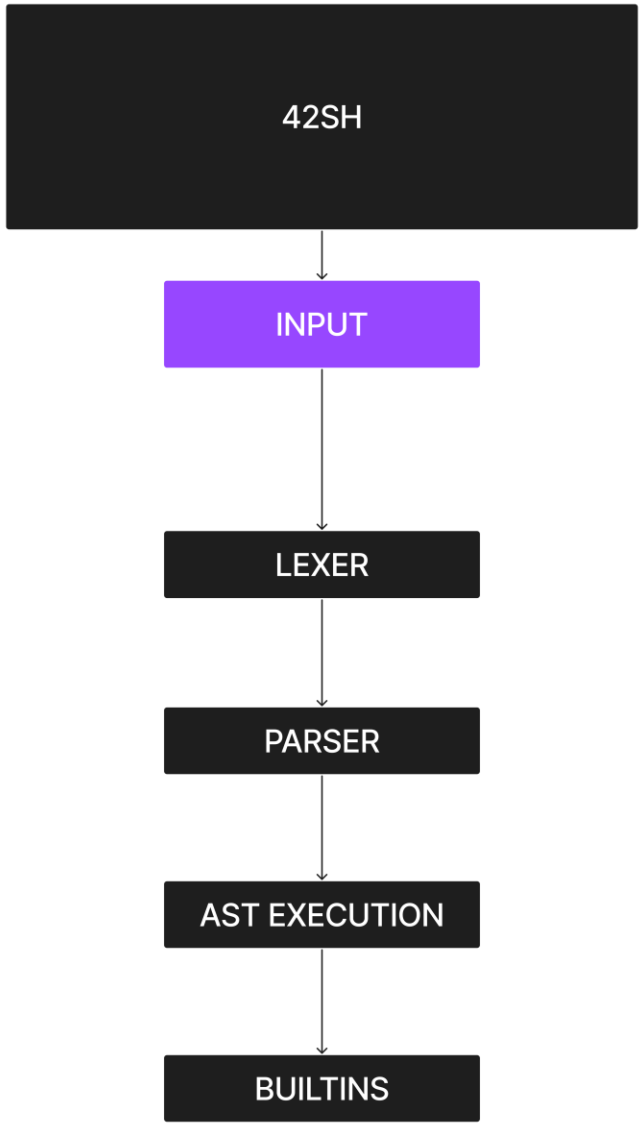
4:SH

- Sommaire:
    - Input
    - Lexer
    - Parser
    - Exécution AST
    - Builtins



INPUT

- **<u>Definition:</u> Premier élément de comprehension de la saisie sous plusieurs formes**



INPUT

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
✦ ➜  ./src/42sh -c "echo Hello World" --pretty-print
Tokens:
Type: 17, Value: echo
Type: 17, Value: Hello
Type: 17, Value: World
Type: 37, Value: EOF


                                0: shell
                                  0: echo Hello World


Hello World
```

INPUT

```
vim script.sh
1 echo Hello World
2
3
~
~
~
```

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
✦ ➜  ./src/42sh script.sh --pretty-print
Tokens:
Type: 17, Value: echo
Type: 17, Value: Hello
Type: 17, Value: World
Type: 37, Value: EOF


                                0: shell
                                  0: echo Hello World


Hello World
```

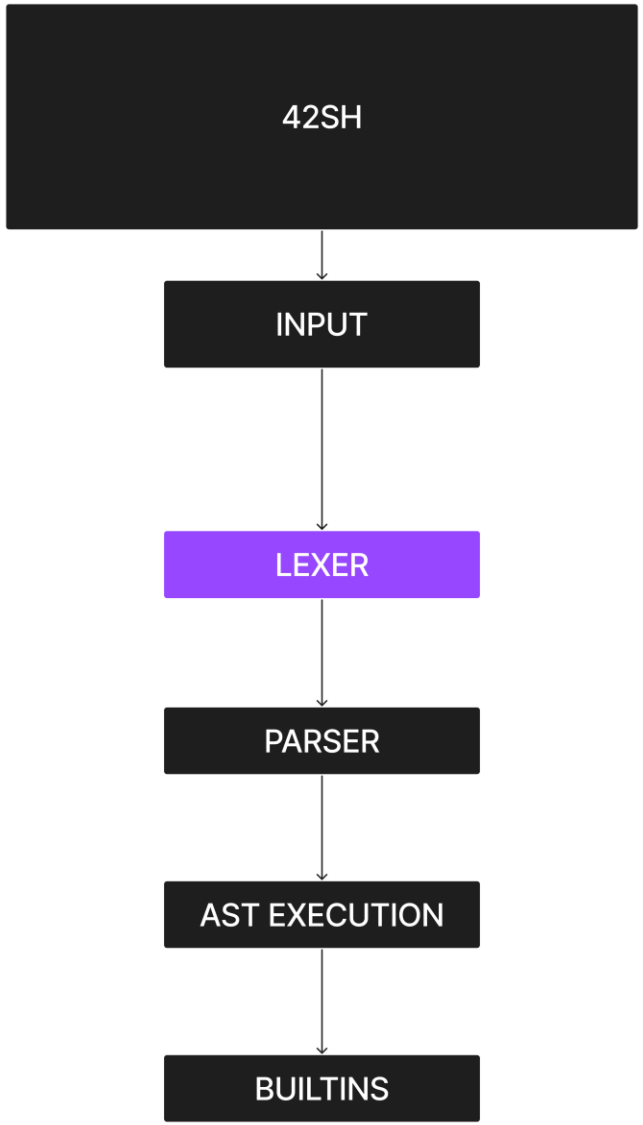
INPUT

```
vim script.sh
1 echo Hello World
2
3
~
~
~
```

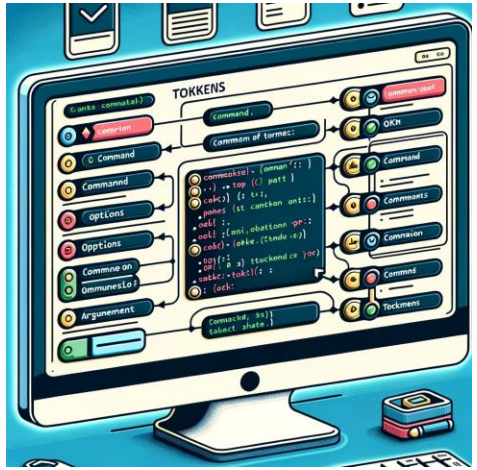
```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
✦ ➜  ./src/42sh < script.sh --pretty-print
Tokens:
Type: 17, Value: echo
Type: 17, Value: Hello
Type: 17, Value: World
Type: 15, Value:

Type: 37, Value: EOF


                                0: shell
                                  0: echo Hello World


Hello World
```

Lexer

- **<u>Definition</u>: Un lexer, ou analyseur lexical, a pour but de diviser le texte d'entrée en une séquence de tokens, qui sont les éléments constitutifs du langage analysé.**



TOKEN



- **Structure des tokens:**
  - **-TYPE**
  - **-VALEUR**

Exemple :

Hello -> TOKEN_WORD

"Hello"

```
enum TokenType
{
    TOKEN_IF,
    TOKEN_THEN,
    TOKEN_ELIF,
    TOKEN_ELSE,
    TOKEN_FI,
    TOKEN_WHILE,
    TOKEN_DO,
    TOKEN_DONE,
    TOKEN_UNTIL,
    TOKEN_FOR,
    TOKEN_IN,
    TOKEN_CASE,
    TOKEN_ESAC,
    TOKEN_SEMICOLON,
    TOKEN_D_SEMICOLON,
    TOKEN_NEWLINE,
    TOKEN_QUOTE,
    TOKEN_WORD,
    TOKEN_VAR,
    TOKEN_ASSIGNMENT_WORD,
    TOKEN_NEG,
    TOKEN_PIPE,
    TOKEN_AND,
    TOKEN_OR,
    TOKEN_IONUMBER,
    TOKEN_GREATER,
    TOKEN_LESS,
    TOKEN_DOUBLE_GREATER,
    TOKEN_GREATER_AMPERSAND,
    TOKEN_LESS_AMPERSAND,
    TOKEN_GREATER_PIPE,
    TOKEN_LESS_GREATER,
    TOKEN_CONCAT,
    TOKEN_OPEN_BRACKETS,
    TOKEN_CLOSE_BRACKETS,
    TOKEN_OPEN_PARENTHESIS,
    TOKEN_CLOSE_PARENTHESIS,
    TOKEN_EOF
};
```

```
struct token
{
    enum TokenType type;
    char *value;
};

struct tokenVect
{
    struct token **data;
    size_t len;
    size_t pos;
    size_t capacity;
};
```

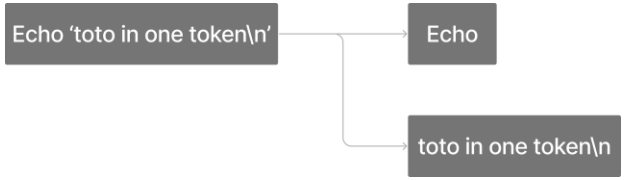
Lexer

- **Garder le contexte en memoire.**
- **Analyser le mot une fois fini.**

```
struct lexing_param
{
    struct tokenVect *tokens;
    char *word;
    size_t size_word;
    size_t word_capacity;
    bool in_squote;
    bool in_dquote;
    bool was_squoted;
    bool was_dquoted;
    bool in_comment;
    bool in_backslash;
    bool was_backslash;
    bool is_assignment_word;
};
```

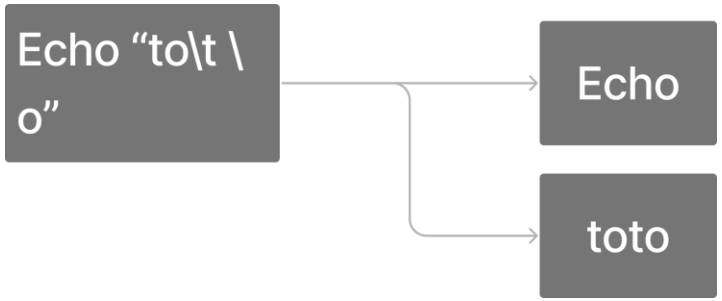
Les fonctionnalités

Simple guillemets: Garde le texte brut.

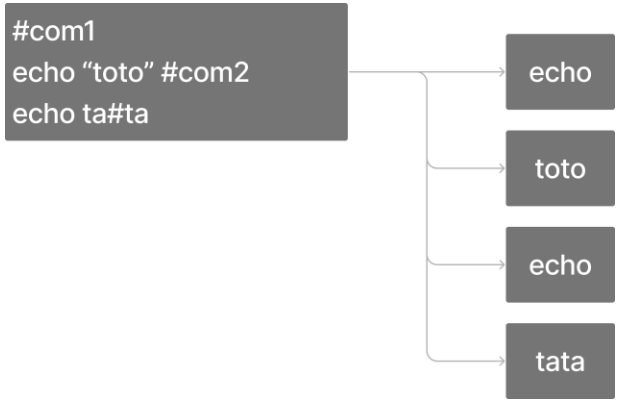


Double guillemets: Étends les backquotes, $ et \

Slash inverse: Garde la valeur brute de la lettre derrière ou colle deux parties.



Commentaire: Les commentaires ne doivent pas être pris en compte, donc pas de token pour eux.

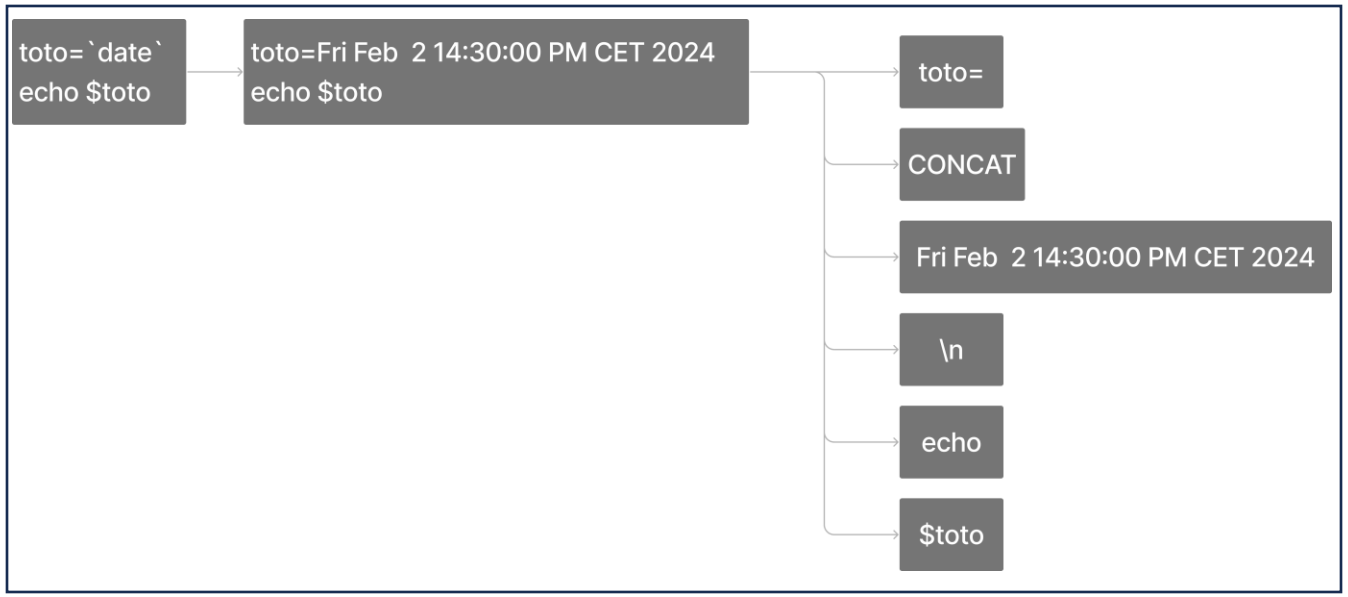


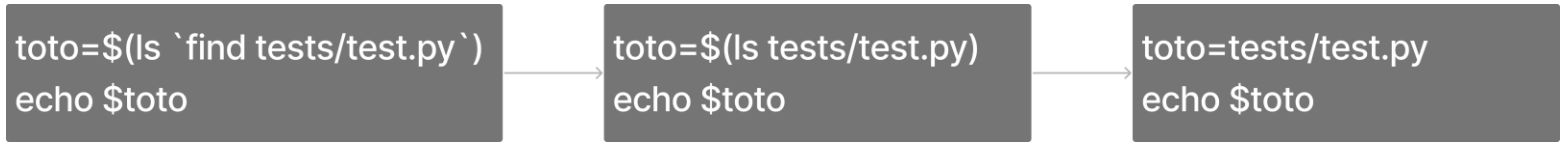
Test

```
./src/42sh script.sh
toto
ta#ta
```

Les substitutions de commandes

Caractère à reconnaître: `` et $()

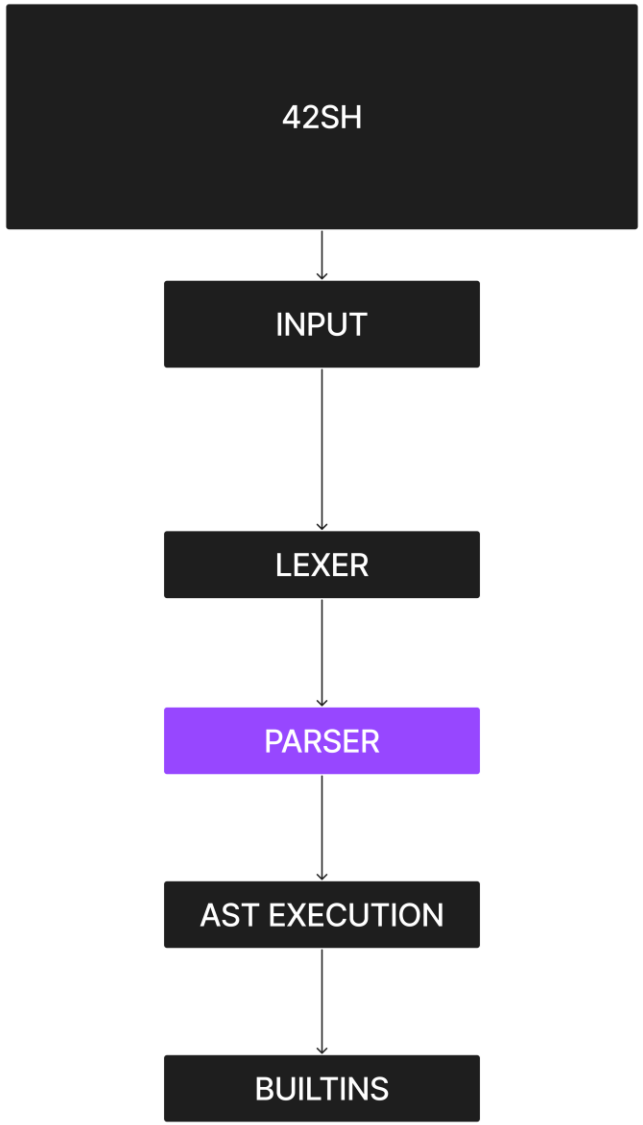





```
./src/42sh script.sh --pretty-print
Tokens:
Type: 19, Value: toto=
Type: 32, Value: CONCAT
Type: 19, Value: tests/test.py
Type: 15, Value:

Type: 17, Value: echo
Type: 18, Value: $toto
Type: 37, Value: EOF
```

Parser

- **<u>Définition:</u> Parcourir le contenu d'un texte ou d'un fichier en l'analysant pour vérifier sa syntaxe ou en extraire des éléments**



Une grammaire récursive

- Cette grammaire permet de reconnaître les fonctions:

$$f(x, g(y))$$

$$
\begin{array}{lcl}
S & \rightarrow & FA \\
 & | & V \\
F & \rightarrow & f \\
 & | & g \\
A & \rightarrow & (S) \\
 & | & (S, S) \\
V & \rightarrow & x \\
 & | & y
\end{array}
$$

Concrètement dans 42sh

- Le parseur permet de vérifier si l'entrée de l'utilisateur respecte les règles de grammaire d'un interpréteur de commandes:

```
simple_command = WORD { element } ;
element = WORD ;
```

```
❯ ./src/42sh -c "echo toto";
Lecture d'une commande simple.
Lecture d'un mot.
Lecture d'un element.

toto
```

```
pipeline = command { '|' {'\n'} command } ;
```

```
❯ ./src/42sh -c "echo toto | cat -e"
Lecture d'une commande simple.
Lecture d'un mot.
Lecture d'un element.
Lecture d'un pipe.
Lecture d'une commande simple.
Lecture d'un mot.
Lecture d'un element.

toto$
```

Un plus grand aperçu

```
input =
    list '\n'
  | list EOF
  | '\n'
  | EOF
  ;

list = and_or { ( ';' | '&' ) and_or } [ ';' | '&' ] ;

and_or = pipeline { ( '&&' | '||' ) {'\n'} pipeline } ;

pipeline = ['!'] command { '|' {'\n'} command } ;

command =
    simple_command
  | shell_command  { redirection }
  | funcdec { redirection }
  ;

simple_command =
    prefix { prefix }
  | { prefix } WORD { element }
  ;

shell_command =
    '{' compound_list '}'
  | '(' compound_list ')'
  | rule_for
  | rule_while
  | rule_until
  | rule_case
  | rule_if
  ;

funcdec = WORD '(' ')' {'\n'} shell_command ;
```

Mais pas de panique

La grammaire se construit étape par étape, en complexifiant chaque règle au fur et à mesure.

```
list = and_or ;
and_or = pipeline ;
pipeline = command ;
command = simple_command ;
```

```
list = and_or { ';' and_or } [ ';' ] ;
and_or = pipeline ;
pipeline = command ;
command = simple_command ;
```

```
pipeline = command { '|' {'\n'} command } ;
```

```
pipeline = ['!'] command { '|' {'\n'} command } ;
```

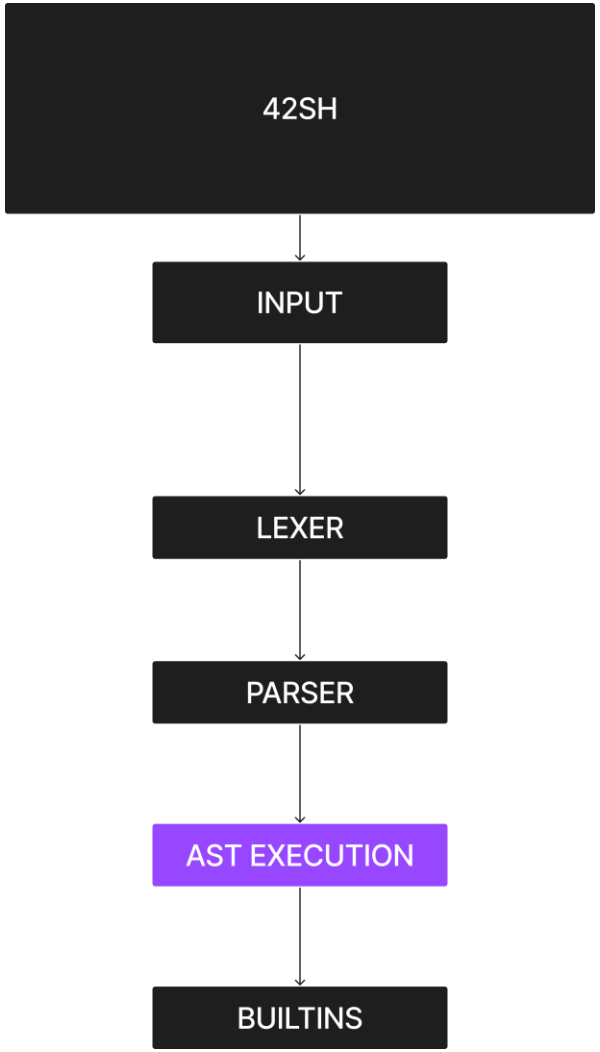
Construction d'un arbre

- À chaque étape de la récursion, on crée un arbre syntaxique.

```
❯ ./src/42sh -c "echo toto"
Arbre de départ.
    0: shell
Ajout du noeud commande
    0: shell
      0: echo
Ajout des arguments de la commande
    0: shell
      0: echo toto
Arbre d'arrivée.
    0: shell
      0: echo toto
toto
```

```
❯ ./src/42sh -c "if true; then echo toto; else echo tata; fi"
  0: shell
    1: if
      0: shell
        0: true
      0: shell
        0: echo toto
      0: shell
        0: echo tata
toto
```

Exécution AST



Définition:

AST: Un Abstract Syntax Tree est une représentation arborescente simplifiée de la structure syntaxique d'un programme informatique.

Exécution AST: Parcours récursif de l'arbre en exécutant les fonctions associer à chacun de ses nœuds

Exécution AST

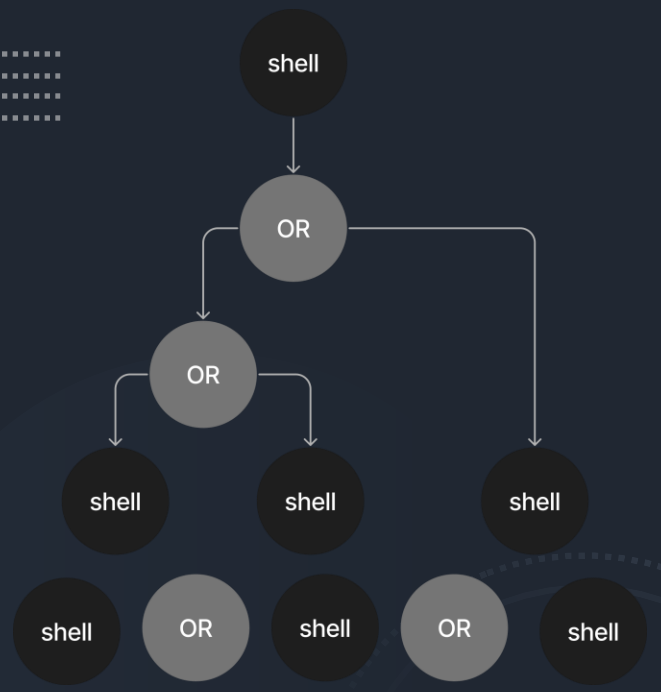
- **AST**
- **Structure des noeuds:**
  - **Commande/Fonction du noeud**
  - **Type du noeud**
  - **Enfants du noeud**
  - **Nombre d'enfants**

```c
struct Command
{
    char *name;
    char *backup;
    struct Command *next;
    enum CommandType type;
};
```

```c
enum CommandType
{
    CMD_WORD,
    CMD_VAR,
    CMD_CONCAT,
    CMD_FREE
};
```

```c
enum ast_type
{
    AST_CMD,
    AST_IF,
    AST_LIST,
    AST_PIPE,
    AST_REDIR,
    AST_NEG,
    AST_OR,
    AST_FOR,
    AST_AND,
    AST_WHILE,
    AST_UNTIL,
    AST_VAR,
    AST_SUB,
    AST_EXP,
    AST_EXIT,
    AST_UNSET,
    AST_DOT,
    AST_FONCTION,
    AST_CASE,
    PARAM,
    PARAM_VAR,
    PARAM_CONCAT
};
```

```c
struct ast
{
    enum ast_type type; // type of the node
    size_t nb_children; // size of children
    struct ast **children; // array of children
    struct Command *command; // data contained in the node
};
```




```c
struct ast *create_node(enum ast_type type, char *data);
struct ast *addNode(struct ast *parent, enum ast_type type, char *cmd);
struct ast *addRootNode(struct ast *oldRoot, enum ast_type type, char *cmd);
void free_command(struct Command *cmd);
void free_ast_node(struct ast *node);
void free_ast(struct ast *root);
struct ast *child_to_root(struct ast *root, struct ast *child);
struct ast *add_parent(struct ast *parent, enum ast_type tipe, char *cmd);
void add_child(struct ast *parent, struct ast *child);
void delete_child(struct ast *parent, size_t index);
```

Exécution AST

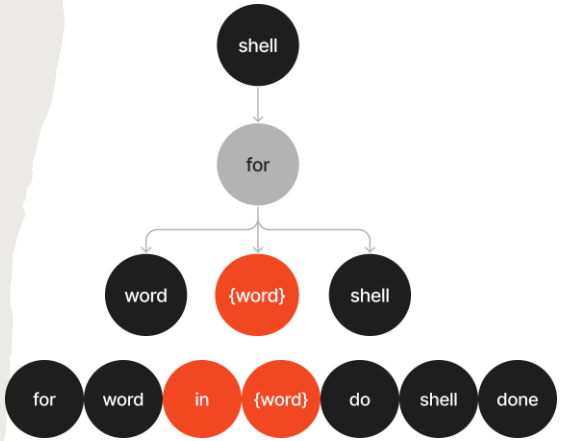
- **Fonctions pour créer l'arbre:**
  - **Création**
  - **Ajouts (Différentes façons)**
  - **Suppression**

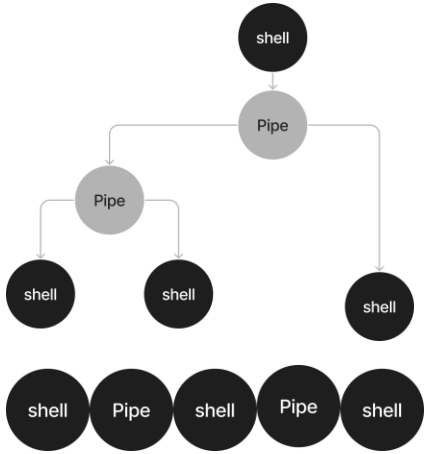
Exécution AST

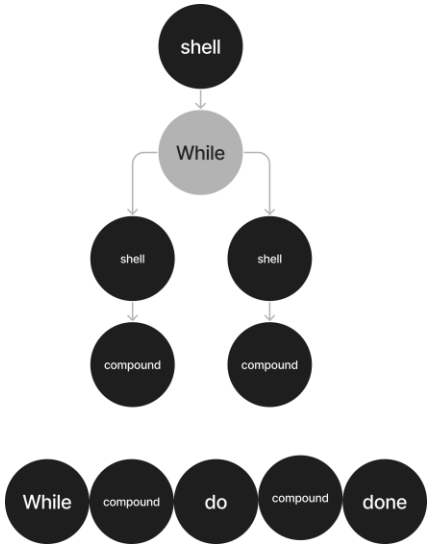
- **Déroulement de l'exécution**
  - **Logique d'exécution (figma)**
  - **Implémentation**
  - **Tests**

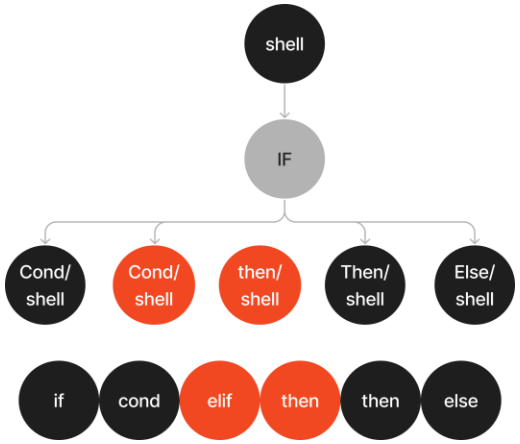
```c
void print_ast(struct ast *node, int depth);
int eval_if(struct ast *node);
int eval_shell(struct ast *node);
int eval_zombie(struct Command *cmd);
int evaluate_node(struct ast *node);
int evaluate_node2(struct ast *node, struct Command *cmd, int return_value);
int evaluate_ast(struct ast *root);
```

```c
if (node->type == AST_VAR)
    return_value = var_builtin(cmd);
else if (strcmp(cmd->name, "continue") == 0)
    return_value = continue_builtin(node);
else if (strcmp(cmd->name, "break") == 0)
    return_value = break_builtin(node);
else if (node->type == AST_PIPE)
    return_value = pipe_builtin(node->children[0], node->children[1]);
else if (node->type == AST_REDIR)
    return_value = redir_builtin(node);
```

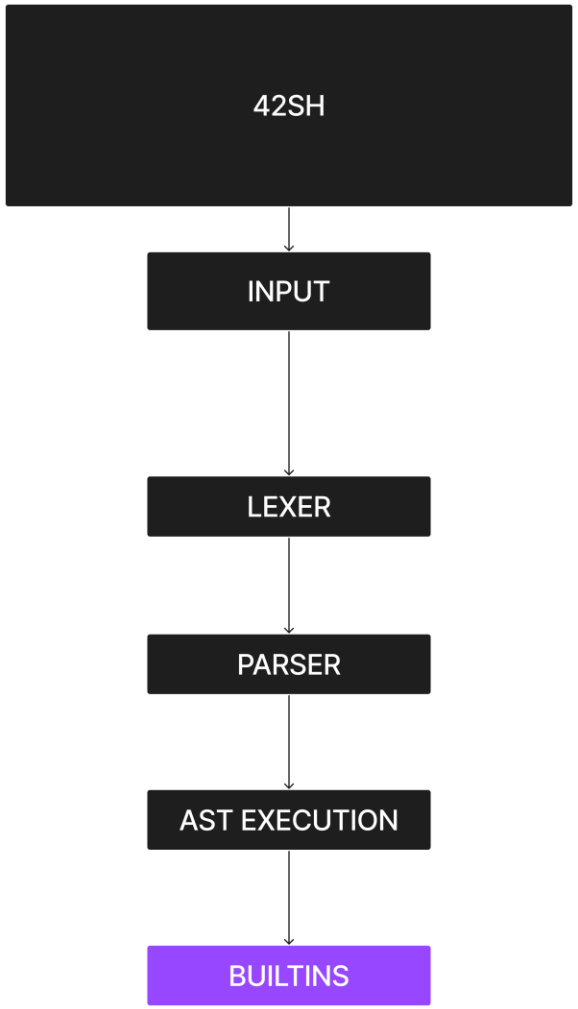








BUILTINS



<u>Définition:</u> Dernière étape avant la sortie standard. Permet l'exécution des commandes de l'AST

BUILTINS: echo

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "echo Hello World"
Hello World
```

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "echo -n Hello World"
Hello World%
```

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "echo -E Hello\\\nWorld"
Hello\nWorld
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "echo -e Hello\\\nWorld"
Hello
World
```

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on ⎇ maste
→  ./src/42sh -c "echo -n -EE -Ene Hello\\\nWorld"
Hello
World
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on ⎇ maste
→  ./src/42sh -c "echo -n -EbE -Ene Hello\\\nWorld"
-EbE -Ene Hello\nWorld%
```

BUILTINS: true, false and exit

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "true"
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  echo $?
0
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "false"
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  echo $?
1
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "exit"
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  echo $?
0
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  ./src/42sh -c "exit 42"
kahel in epita-ing-assistants-acu-42sh-2026-paris-65
→  echo $?
42
```

BUILTINS: cd

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
➜  ./src/42sh -c "echo \$PWD; cd; echo \$PWD"
/home/kahel/epita/prog/42sh/epita-ing-assistants-acu-42sh-2026-paris-65
/home/kahel
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
➜  ./src/42sh -c "echo \$PWD; cd ..; echo \$PWD; cd -; echo \$PWD"
/home/kahel/epita/prog/42sh/epita-ing-assistants-acu-42sh-2026-paris-65
/home/kahel/epita/prog/42sh
/home/kahel/epita/prog/42sh/epita-ing-assistants-acu-42sh-2026-paris-65
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
➜  ./src/42sh -c "echo \$PWD; cd /tmp;echo \$PWD"
/home/kahel/epita/prog/42sh/epita-ing-assistants-acu-42sh-2026-paris-65
/tmp
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
➜  ./src/42sh -c "echo \$PWD; cd src/ast/;echo \$PWD"
/home/kahel/epita/prog/42sh/epita-ing-assistants-acu-42sh-2026-paris-65
/home/kahel/epita/prog/42sh/epita-ing-assistants-acu-42sh-2026-paris-65/src/ast
```

BUILTINS: redirections

```
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
→  ls
aclocal.m4  autom4te.cache  compile        configure     install-sh   Makefile.in  NEWS       script.sh
ar-lib      ChangeLog       config.log     configure.ac  Makefile     make.sh      README     src
AUTHORS     clear.sh        config.status  depcomp       Makefile.am  missing      README.md  tests
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
→  ./src/42sh -c "echo Redirection Worked > file"
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
→  ls
aclocal.m4  autom4te.cache  compile        configure     file         Makefile.am  missing  README.md  tests
ar-lib      ChangeLog       config.log     configure.ac  install-sh   Makefile.in  NEWS     script.sh
AUTHORS     clear.sh        config.status  depcomp       Makefile     make.sh      README   src
kahel in epita-ing-assistants-acu-42sh-2026-paris-65 on master [⇡?]
→  cat file
Redirection Worked
```

Section 3

42sh