Projet ICE4404P

Ce projet s'inscrit dans le cadre du cours ICE 4404P-Théorie des langages de programmation. Terminé par Nan Lin (5212619100008) et YingRui ShangGuan (5212609100006). Achevé le 13 janvier 2025.

Table des matières

1. Avant tout

Nous apprécions beaucoup notre travail mutuel et la collaboration entre nous. Ce projet n'aurait jamais vu le jour sans chacun d'entre nous.

Le code est écrit en anglais. Aucun d'entre nous n'est bon en français, c'est pourquoi nous écrivons d'abord le code et les notations en anglais, puis nous les traduisons en français.

Pour exécuter ce programme :

```
# Assure that you're currently in the root directory of this folder
make
```

Vous observerez une séquence de sorties:

```
mkdir -p build
gcc -g -Wall -Wextra -c src/lex/analex.c -o build/analex.o
gcc -g -Wall -Wextra -c src/lex/test_analex.c -o build/test_analex.o
gcc -g -Wall -Wextra -o build/test_analex build/analex.o build/test_analex.o
gcc -g -Wall -Wextra -c src/syn/anasynt.c -o build/anasynt.o
gcc -g -Wall -Wextra -c src/syn/test_anasynt.c -o build/test_anasynt.o
gcc -g -Wall -Wextra -o build/test_anasynt build/anasynt.o build/test_anasynt.o build/
analex.o
gcc -g -Wall -Wextra -c src/sem/anasem.c -o build/anasem.o
gcc -g -Wall -Wextra -c src/sem/test_anasem.c -o build/test_anasem.o
gcc -g -Wall -Wextra -o build/test_anasem build/anasem.o build/test_anasem.o build/
anasynt.o build/analex.o
gcc -g -Wall -Wextra -c src/compil/compilateur.c -o build/compilateur.o
gcc -g -Wall -Wextra -c src/compil/test_compilateur.c -o build/test_compilateur.o
gcc -g -Wall -Wextra -c src/runtime/runtime.c -o build/runtime.o
gcc -g -Wall -Wextra -o build/test_compilateur build/compilateur.o build/
test_compilateur.o build/analex.o build/anasynt.o build/anasem.o build/runtime.o
gcc -g -Wall -Wextra -c src/runtime/test_runtime.c -o build/test_runtime.o
gcc -g -Wall -Wextra -o build/test_runtime build/runtime.o build/test_runtime.o build/
compilateur.o build/analex.o build/anasynt.o build/anasem.o
```

Aucun avertissement n'existe dans le code ! Nous sommes très fiers de nous ;)

Après avoir construit les exécutables, pour testez les différentes parties :

```
# L'analyseur lexical
build/test_analex testfiles/Regles.txt

# L'analyseur syntaxique
build/test_anasynt testfiles/Regles.txt

# L'analyseur sémantique
build/test_anasem testfiles/Regles.txt

# La compilateur
build/test_compilateur testfiles/Regles.txt

# La VM
build/test_runtime testfiles/Init.txt
```

2. L'analyseur lexcial

Task 1 — L'analyseur lexcial

On produira

1. *Un programme C, effectuant l'analyseur lexical dans un fichier* `analex.c` *;*
2. *Un programme de test* `test_analex.c`*, contenant une fonction* `main` *et permettant de tester sur une proposition le bon comportement de l'analyseur lexical. Ce programme d'analyse lexicale doit être capable de détecter des erreurs lexicales ! S'il n'y a pas d'erreur, le programme devra produire une liste (ou un tableau) de lexèmes, il sera donc de type* `lexeme list analyseur_lexical(char *)` *;*
3. *Une explication du code (expressions régulières, automates, choix effectués, problèmes rencontrés, comment ils ont été résolus, etc.) dans le rapport.*

Puisque notre entrée contient des lettres spéciales Unicode, nous devons les remplacer par des alphabets standardisés.

```
// analex.h
// Define UTF-8 Characters
typedef struct {
    char *name;
    char unicode[4];
    int size;
} Utf8Char;

Utf8Char UTF8_CHARS[] = {
    {"IMPLIQUE", {(char)0xE2, (char)0x87, (char)0x92}, 3}, // ⇒
    {"PRODUIT", {(char)0xE2, (char)0x86, (char)0x92}, 3},  // →
    {"NON", {(char)0xC2, (char)0xAC}, 2},                  // ¬
    {"ET", {(char)0xE2, (char)0x88, (char)0xA7}, 3},       // ∧
    {"OU", {(char)0xE2, (char)0x88, (char)0xA8}, 3}        // ∨
};
```

Cela définit la correspondance entre la variable de chaîne transformée et le codage Unicode d'origine. Nous enregistrons également le nombre de valeurs `char` auxquelles chaque caractère Unicode correspond pour la détection dans une séquence d'entrées de caractères.

Les types de *tokens* de base sont les suivants :

- la parenthèse gauche et la parenthèse droite
- Les opérateurs
- la variable propositionnelle
- Le « produit » (qui est le lien entre deux propositions)

Ainsi, nous définissons le `token_type` comme :

```
typedef enum {
    PO,             // '('
    PF,             // ')'
    OP,             // '¬', '∧', '∨', '⇒'
    PROP,           // e.g. "p21"
```

```
    PRODUIT         // "→"
} token_type;
```

Nous définissons ensuite un `lexème` comme une combinaison du `token_type` de base et de son contenu.

```
// lexeme definition
typedef struct {
    token_type type; // e.g. Prop
    char value[10];  // e.g. "p2" or NULL
} lexeme_;
typedef lexeme_ *lexeme;
```

Le résultat attendu est une liste de ces lexèmes. De plus, il doit enregistrer la longueur du lexème et s'il est valide ou non :

```
// lexeme list definition
#define MAX_LEXMEMES 50
typedef struct {
    lexeme lexemes[MAX_LEXMEMES]; // a list of lexemes
    int size;                     // number of lexemes contained in the list
    bool error;                   // = 1 indicate exists error
} lexeme_list;
```

Le type `bool` est défini à l'avance comme un type booléen.

Le programme `test_analex` reçoit en entrée le nom du fichier à lire. La première partie de la fonction principale est la suivante

```
// test_analex.c
#include "analex.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: ./test_analex <FILENAME>\n");
        return 1;
    }

    const char *filename = argv[1];
    FILE *fp = fopen(filename, "r");
    if (!fp) {
        fprintf(stderr, "Cannot open file %s\n", filename);
        return 1;
    }

  // ...
}
```

En utilisant les définitions des tokens, des lexèmes et des listes de lexèmes, le flux de travail pour séparer les chaînes d'entrée de ligne en listes de lexèmes est listé ci-dessous :

```
// test_analex.c

int main(int argc, char *argv[]) {
  // ...
```

```
    char line[256]; // suppose max number of line could be read is 256
    int line_number = 1;

    // Deal with each line
    while(fgets(line, sizeof(line), fp)) {
        lexeme_list result = analyseur_lexical(line);

        printf("Line %d: %s", line_number, line);
        printf("Lexical analysis result: ");
        print_lexeme_list(result);
        printf("\n\n");

        line_number++;
    }

    return 0;
}
```

Pour générer le résultat attendu, nous avons également besoin d'une fonction `print_lexeme_list`.

Cette fonction est relativement simple :

- Tout d'abord, nous vérifions si le drapeau d'erreur est vrai ou faux.
- Si aucune erreur n'est détectée, nous imprimons chaque lexème de la liste en utilisant une instruction switch-case et une boucle for.

```
// analex.c
// Print to stdout stream
void print_lexeme_list(const lexeme_list result) {
    if (result->error) {
        fprintf(stderr, "Lexical analysis has failed.\n");
        return;
    }

    printf("[");
    // For every lexeme in the lexeme list
    for (int i = 0; i < result->size; i++) {
        switch (result->lexemes[i]->type) {
            case PO:
                printf("PO, ");
                break;
            case PF:
                printf("PF, ");
                break;
            case OP:
                printf("Op(\"%s\"), ", result->lexemes[i]->value);
                break;
            case PROP:
                printf("Prop(\"%s\"), ", result->lexemes[i]->value);
                break;
            case PRODUIT:
                printf("PRODUIT, ");
                break;
```

```
            default:
                fprintf(stderr, "Unrecognized lexeme type.\n");
                return;
        }
    }
    printf("]\n");
}
```

La première étape consiste à initialiser la variable de sortie `lexeme_list`. Nous la nommons `result`.

```
// analex.c
lexeme_list analyseur_lexical(const char *line_str) {
    lexeme_list result = (lexeme_list)malloc(sizeof(lexeme_list_));
    if (!result) {
        fprintf(stderr, "Memory allocation error in creating lexeme list.\n");
        return NULL;
    }

    result->size = 0;
    result->error = false;
  // ...
}
```

Ensuite, nous parcourons la chaîne de lignes d'entrée.

- Chaque fois que nous détectons un lexème valide, nous créons une « instance » de lexème et l'ajoutons à la liste.
- Les espaces blancs sont ignorés et supprimés lorsqu'ils sont détectés.
- Le processus de détection comprend l'identification des propositions (une combinaison de chiffres et d'alphabets) et des caractères Unicode (composés de 2 ou 3 variables `char`).

```
// analex.c
// The basic types
// Auxiliary function to help detect whitespace in the string input
static bool is_whitespace(char c) {
    return (c == ' ' || c == '\t' || c == '\n' || c == '\r');
}

static bool is_alpha(char c) {
    return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
}

static bool is_digit(char c) {
    return (c >= '0' && c <= '9');
}
```

Puisque la création d'une « instance » de lexème implique `malloc` et la copie de chaîne, nous définissons une fonction auxiliaire nommée `create_lexeme`.

```
// Auxiliary function to help create a lexeme "instance"
static lexeme create_lexeme(token_type t, const char *val) {
    lexeme new_lex = (lexeme)malloc(sizeof(lexeme_));
    if (!new_lex) {
        fprintf(stderr, "Memory allocation error in creating new lexeme.\n");
```

```
        return NULL;
    }
    new_lex->type = t;
    if (val) {
        strncpy(new_lex->value, val, sizeof(new_lex->value) - 1);
        new_lex->value[sizeof(new_lex->value) - 1] = '\0';
    } else {
        new_lex->value[0] = '\0';
    }
    return new_lex;
}
```

Sur la base des fonctions ci-dessus, nous implémentons la première partie de la fonction `create_lexeme`, qui peut détecter les lexèmes constitués d'un seul caractère (par exemple, les parenthèses gauches et les parenthèses droites).

```
// analex.c
lexeme_list analyseur_lexical(const char *line_str) {
  // ...
  int i = 0, length = strlen(line_str);

  while (i < length) {
        if (is_whitespace(line_str[i])) {
            i++;
            continue;
        }

        if (result->size >= MAX_LEXMEMES) {
            fprintf(stderr, "[Lexical Error] Too many tokens.\n");
            result->error = true;
            return result;
        }

        if (line_str[i] == '(') {
            result->lexemes[result->size++] = create_lexeme(PO, "(");
            i++;
            continue;
        }

        if (line_str[i] == ')') {
            result->lexemes[result->size++] = create_lexeme(PF, ")");
            i++;
            continue;
        }

    // ...
    }
}
```

Ensuite, nous définissons la partie qui peut détecter les caractères Unicode. Nous définissons la fonction `match_utf8_char` pour :

- Se déplacer vers un `char` dans la séquence d'entrée et déterminer s'il correspond à un élément du dictionnaire `Utf8Char` enregistré ci-dessus.

- Comparer les caractères et identifier le lexème qui correspond à l'enregistrement `Utf8Char`.

```
// analex.c
static int match_utf8_char(const char *str, Utf8Char *matched_char) {
    for (int i = 0; i < NUM_UTF8_CHARS; i++) {
        if (strncmp(str, UTF8_CHARS[i].unicode, UTF8_CHARS[i].size) == 0) {
            if (matched_char) {
                *matched_char = UTF8_CHARS[i];
            }
            return UTF8_CHARS[i].size;
        }
    }
    return 0;
}

lexeme_list analyseur_lexical(const char *line_str) {
  // ...

  while (i < length) {
    // ...
    Utf8Char matched_char;
        int utf8_len = match_utf8_char(&line_str[i], &matched_char);
        if (utf8_len > 0) {
            token_type type;
            if (strcmp(matched_char.name, "IMPLIQUE") == 0) {
                type = OP;
            } else if (strcmp(matched_char.name, "PRODUIT") == 0) {
                type = PRODUIT;
            } else if (strcmp(matched_char.name, "NON") == 0) {
                type = OP;
            } else if (strcmp(matched_char.name, "ET") == 0) {
                type = OP;
            } else if (strcmp(matched_char.name, "OU") == 0) {
                type = OP;
            } else {
                fprintf(stderr, "[Lexical Error] Unrecognized UTF-8 character.\n");
                result->error = true;
                return result;
            }
            result->lexemes[result->size++] = create_lexeme(type, matched_char.name);
            i += utf8_len;
            continue;
  }

  // ...
}
```

Enfin, nous détectons les propositions dont la longueur variable est inconnue. Pour ce faire, nous définissons un tampon `int` pour stocker tous les caractères qui satisfont la regex `[alpha][digits]*` jusqu'à ce qu'elle ne corresponde plus.

```
// analex.c
lexeme_list analyseur_lexical(const char *line_str) {
  // ...
```

```
  char buffer[10];
  int buffer_idx = 0;
  while (i < length) {
    // ...
    if (is_alpha(line_str[i])) {
            buffer_idx = 0;
            while (is_alpha(line_str[i]) || is_digit(line_str[i])) {
                if (buffer_idx < (int)sizeof(buffer) - 1) {
                    buffer[buffer_idx++] = line_str[i];
                }
                i++;
            }
            buffer[buffer_idx] = '\0';
            result->lexemes[result->size++] = create_lexeme(PROP, buffer);
            continue;
        }
    // ...
  }
}
```

Enfin, nous gérons le cas où aucun lexème ne correspond. Nous retournons la `lexeme_list` en sortie.

```
lexeme_list analyseur_lexical(const char *line_str) {
  // ...
  while (i < length) {
    // ...
        fprintf(stderr, "[Lexical Error] Unexpected character: '%c'\n", line_str[i]);
        result->error = true;
        return result;
    }

    return result;
}
```

Problèmes rencontrés:

1. Problèmes d'assignation de pointeurs et de malloc : Nous avons créé une fonction auxiliaire pour gérer la création de la structure `lexeme` et de la structure `lexeme_list`. De plus, nous avons besoin de copier la chaîne de caractères, donc nous utilisons la fonction `strncpy` pour atteindre le but.
2. Les caractères Unicode qui traitent des problèmes : ils occupent trois caractères au lieu d'un. Nous avons donc besoin de méthodes spéciales pour lire ces caractères. Nous ne pouvons pas simplement comparer les lettres Unicode à une variable char.

   ```
   src/lex/analex.c:69:28: error: character too large for enclosing character literal
   type
      69 |         if (line_str[i] == '¬' || line_str[i] == '∧' || line_str[i] == '∨'
   || line_str[i] == '⇒') {
   ```

3. Un *buffer* supplémentaire est conçu pour détecter la proposition.

Exécution:


PROBLEMS 5
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
GITLENS
zsh
make build/test_analex
mkdir -p build
gcc -g -Wall -Wextra -c src/lex/analex.c -o build/analex.o
gcc -g -Wall -Wextra -c src/lex/test_analex.c -o build/test_analex.o
gcc -g -Wall -Wextra -o build/test_analex build/analex.o build/test_analex.o
build/test_analex testFiles/Regles.txt
Line 1: (p1⇒p2)⇔(¬p1∨p2)
Lexical analysis result: [PO, Prop("p1"), Op("IMPLIQUE"), Prop("p2"), PF, PRODUIT, PO, PO, Op("NON"), Prop("p1"), PF, Op("OU"), Prop("p2"), PF,]
Line 2: (p1∨(p2∧p3))⇔((p1∨p2)∧(p1∨p3))
Lexical analysis result: [PO, Prop("p1"), Op("OU"), PO, Prop("p2"), Op("ET"), Prop("p3"), PF, PF, PRODUIT, PO, PO, Prop("p1"), Op("OU"), Prop("p2"), PF, Op("ET"), PO, Prop("p1"), Op("OU"), Prop("p3"), PF, PF,]
Line 3: (p1∨p2)⇒(p2∨p1)
Lexical analysis result: [PO, Prop("p1"), Op("OU"), Prop("p2"), PF, PRODUIT, PO, Prop("p2"), Op("OU"), Prop("p1"), PF,]
Line 4: (p1∧p2)⇒(p2∧p1)
Lexical analysis result: [PO, Prop("p1"), Op("ET"), Prop("p2"), PF, PRODUIT, PO, Prop("p2"), Op("ET"), Prop("p1"), PF,]
Line 5: (¬(p1∧p2))⇔((¬p1)∨(¬p2))
Lexical analysis result: [PO, Op("NON"), PO, Prop("p1"), Op("ET"), Prop("p2"), PF, PF, PRODUIT, PO, PO, Op("NON"), Prop("p1"), PF, Op("OU"), PO, Op("NON"), Prop("p2"), PF, PF,]
Line 6: (¬(p1∨p2))⇔((¬p1)∧(¬p2))
Lexical analysis result: [PO, Op("NON"), PO, Prop("p1"), Op("OU"), Prop("p2"), PF, PF, PRODUIT, PO, PO, Op("NON"), Prop("p1"), PF, Op("ET"), PO, Op("NON"), Prop("p2"), PF, PF,]
Line 7: (¬(¬p1))⇒p1
Lexical analysis result: [PO, Op("NON"), PO, Op("NON"), Prop("p1"), PF, PF, PRODUIT, Prop("p1"),]
~/L/Mo/com-apple-C/Doc/S/C/24-a/ICE4404/Compilers-Project-2025 on semantic-analyzer +27 !1
at 18:36:15

3. L'analyseur syntaxique

Task 2 — L'analyseur syntaxique

On produira

1. *Un programme C, effectuant l'analyseur syntaxique dans un fichier* `anasynt.c` *;*
2. *Un programme de test* `test_anasynt.c`*, contenant une fonction* `main` *et permettant de tester sur une proposition le bon comportement de l'analyseur syntaxique. Ce programme d'analyse syntaxique doit être capable de détecter des erreurs syntaxiques ! S'il n'y a pas d'erreur, le programme devra produire un arbre syntaxique, il sera donc de type* `arbre_syntaxique analyseur_lexical(lexeme list)` *;*
3. *Une explication du code (grammaire BNF, automates à pile, choix effectués, problèmes rencontrés, comment ils ont été résolus, etc.) dans le rapport.*

Notre *BNF* grammaire est définie comme suit :

```
<S> ::= <Formula>

<Formula> ::= <Implication>
            | <Implication> PRODUIT <Implication>

<Implication> ::= <Disjunction>
                | <Disjunction> OP(⇒) <Disjunction>

<Disjunction> ::= <Conjunction>
                | <Disjunction> OP(∨) <Conjunction>

<Conjunction> ::= <Negation>
                | <Conjunction> OP(∧) <Negation>

<Negation> ::= OP(¬) <Negation>
             | PO <Formula> PF
             | PROP
```

Chaque nœud de l'AST est représenté par la structure suivante. Nous distinguons trois types de nœuds principaux :

- `AST_PROP` pour les propositions
- `AST_OP` pour le sopérateurs logiques
- `AST_PRODUCE` pour →

```c
// AST node type
typedef enum {
    AST_PROP,
    AST_OP,
    AST_PRODUCE
} ast_node_type;

// AST node struct
typedef struct ast_node {
```

```
    ast_node_type type;
    char prop[10];                  // valid if type==AST_PROP
    char operator[10];              // valid if type==AST_OP
    struct ast_node *left;
    struct ast_node *right;
} ast_node;
typedef ast_node* AST;
```

Les pointeurs enfants `left` et `right` peuvent représenter la structure des sous-expressions. Pour les opérateurs unaires (par exemple `NON`), `droite` est souvent `NULL`.

Nous implémentons l'analyse syntaxique *recursive-descent*. Chaque règle de grammaire correspond à une fonction qui consomme des tokens afin de construire des nœuds AST. Par exemple :

- `parseFormula()` correspond à la règle `<Formula>`.
- `parseImplication()` correspond à la règle `<Implication>`, etc.

```
static AST parseFormula();
static AST parseImplication();
static AST parseDisjunction();
static AST parseConjunction();
static AST parseNegation();
```

Il y a deux cas:

- Si la séquence actuelle de *tokens* correspond à la règle de production, nous construisons un noeud d'arbre (ou un sous-arbre) et avançons le pointeur de *token*.
- Si ce n'est pas le cas, nous émettons une erreur et renvoyons `NULL`.

Nous utilisons la variable globale `current_index` pour enregistrer la position où nous nous trouvons dans la liste des lexèmes.

```
// record the pos in the lexeme list
static lexeme_list current_list = NULL;
static int current_index = 0;

static lexeme current_token() {
    if (current_index < current_list->size) {
        return current_list->lexemes[current_index];
    }
    return NULL; // = No tokens left
}
```

Le déroulement de l'analyse lexicale est simple : Nous commençons par détecter les tokens un par un et nous créons le nœud ast correspondant que nous plaçons dans l'arbre en cours de route. Cependant, l'implémentation de chaque règle BNF est difficile.

La fonction ci-dessous est utilisée pour passer au lexème suivant dans la liste (un processus de « consommation » de lexèmes) :

```
static void advance() {
    if (current_index < current_list->size) {
        current_index++;
    }
}
```

La mise en œuvre détaillée de chaque formule BNF est représentée dans les notes et le code de chaque fonction.

Commençons par la structure la plus fondamentale. Selon notre définition, notre résultat final doit être une formule unique, puisque nous définissons la fonction initiale.

```
// <Start> ::= <Formula>
static AST parseStart() {
    // <S> = <Formula>
    return parseFormula();
}
```

La formule peut être définie de deux manières, soit par une seule Implication, soit par deux Implications lorsque le symbole du produit est présent. Nous les mettons donc en correspondance selon les différents cas. En particulier, nous devons détecter les nœuds suivants.

```
// <Formula> ::= <Implication>
//             | <Implication> PRODUIT <Implication>
static AST parseFormula() {
    AST left = parseImplication();
    if (!left) return NULL;

    if (current_token() && current_token()->type == PRODUIT) {
        advance();

        // parse the second implication
        AST right = parseImplication();
        if (!right) return NULL;

        AST node = (AST)malloc(sizeof(ast_node));
        node->type = AST_PRODUCE;
        strcpy(node->operator, "PRODUIT");
        node->prop[0] = '\0';
        node->left = left;
        node->right = right;
        return node;
    }

    return left;
}
```

La définition de l'Implication est très similaire à la première, en ce sens qu'elle peut être une disjonction seule, mais lorsque les symboles de dérivation sont présents, il s'agit d'une combinaison de deux disjonctions qui doivent être appariées en fonction de la situation. Le code est à peu près similaire.

```
// <Implication> ::= <Disjunction>
//                 | <Disjunction> OP(⇒) <Disjunction>
static AST parseImplication() {
    AST left = parseDisjunction();
    if (!left) return NULL;

    if (current_token()
        && current_token()->type == OP
```

```
        && strcmp(current_token()->value, "IMPLIQUE") == 0) {

        lexeme tok = current_token();
        advance();

        AST right = parseDisjunction();
        if (!right) return NULL;

        AST node = (AST)malloc(sizeof(ast_node));
        node->type = AST_OP;
        strcpy(node->operator, tok->value); // =IMPLIQUE
        node->prop[0] = '\0';
        node->left = left;
        node->right = right;

        return node;
    }

    return left;
}
```

La disjonction et la conjonction sont également très similaires et sont omises ici.

Enfin, il y a la définition de NON, qui est divisée en trois cas, premièrement, l'opération NON pour un énoncé de négation est toujours la négation, et ici l'opération NON est toujours la négation dans le premier cas.

```
// <Negation> ::= OP(¬) <Negation>
//              | "(" <Formula> ")"
//              | PROP
static AST parseNegation() {
    lexeme tok = current_token();
    if (!tok) {
        fprintf(stderr, "[Syntax Error] Unexpected end of tokens in parseNegation.\n");
        return NULL;
    }

    if (tok->type == OP && strcmp(tok->value, "NON") == 0) {
        AST node = (AST)malloc(sizeof(ast_node));
        node->type = AST_OP;
        strcpy(node->operator, "NON");
        node->prop[0] = '\0';
        node->left = NULL;
        node->right = NULL;

        advance();

        AST sub = parseNegation();
        if (!sub) {
            free(node);
            return NULL;
        }
        node->left = sub;
```

```
        return node;
    }

    // ...
}
```

S'il détecte une nouvelle proposition, il crée un nouveau nœud, ce qui est le plus simple.

S'il détecte des parenthèses (notez notre implémentation !)), il doit cesser de créer de nouveaux nœuds et revenir en arrière.

```
static AST parseNegation() {
    // ....
    if (tok->type == PO) {
        advance();

        AST inside = parseFormula();
        if (!inside) return NULL;

        if (!current_token() || current_token()->type != PF) {
            fprintf(stderr, "[Syntax Error] Missing closing parenthesis.\n");
            return NULL;
        }
        advance();
        return inside;
    }

    if (tok->type == PROP) {
        AST node = (AST)malloc(sizeof(ast_node));
        node->type = AST_PROP;
        strcpy(node->prop, tok->value);
        node->operator[0] = '\0';
        node->left = NULL;
        node->right = NULL;
        advance();
        return node;
    }

    // ...
}
```

En outre, il convient de noter que dans le processus de test. Nous devons d'abord procéder à une analyse lexicale, puis à une analyse syntaxique.

Pour visualiser le résultat de l'arbre, nous avons défini une fonction print_ast qui permet d'afficher la hiérarchie de la structure AST:

(Nous ajoutons l'implémentation de l'indentation pour donner un aspect plus moderne au texte.)

```
void print_ast(AST node, int indent) {
    if (!node) return;
    for (int i = 0; i < indent; i++) {
        printf("  ");
    }
```

```
    switch (node->type) {
        case AST_PROP:
            printf("PROP(%s)\n", node->prop);
            break;
        case AST_OP:
            printf("OP(%s)\n", node->operator);
            break;
        case AST_PRODUCE:
            printf("PRODUIT\n");
            break;
    }

    print_ast(node->left, indent + 1);
    print_ast(node->right, indent + 1);
}
```

Résultat:

Problèmes rencontrés : La grammaire BNF doit être conçue avec le plus grand soin, sous peine de refléter un grave problème de codage ! Toute petite erreur entraînerait de graves problèmes.

En outre, nous ajoutons quelques cas de test spéciaux et prouvons la fonctionnalité des opérateurs NON, lorsque plusieurs opérateurs NON sont combinés.

```
Line 1: (¬(¬¬¬¬¬¬¬¬¬¬p1))→p1
Syntax tree:
PRODUIT
  OP(NON)
    OP(NON)
      OP(NON)
        OP(NON)
```

```
          OP(NON)
            OP(NON)
              OP(NON)
                OP(NON)
                  OP(NON)
                    OP(NON)
                      OP(NON)
                        OP(NON)
                          OP(NON)
                            PROP(p1)
  PROP(p1)

Line 2: ¬p1vp1
Syntax tree:
OP(OU)
  OP(NON)
    PROP(p1)
  PROP(p1)
```

4. L'analyseur sémantique

Task 3 — L'analyseur sémantique

On produira

1. *Un programme C, effectuant l'analyseur sémantique dans un fichier* `anasem.c` *;*
2. *Un programme de test* `test_anasem.c`*, contenant une fonction* `main` *et permettant de tester sur une proposition le bon comportement de l'analyseur sémantique. Ce programme d'analyse sémantique doit être capable de détecter des erreurs sémantiques ! S'il n'y a pas d'erreur, le programme devra produire un arbre syntaxique correct, il sera donc de type* `arbre_syntaxique analyseur_semantique(arbre_syntaxique)` *;*
3. *Une explication du code (les vérifications effectuées, les problèmes rencontrés, comment ils ont été résolus, etc.) dans le rapport.*

Au cours de cette étape, nous devons vérifier l'exactitude des opérateurs. Plus précisément, les opérateurs NON n'acceptent qu'un seul opérateur, tandis que d'autres en acceptent deux. Il convient ensuite de leur définir un code cfa unique pour le processus d'exécution.

```
// operator struct
typedef struct {
    const char *name;
    int param_count;  // 1 for NOT, 2 for AND/OR/IMPLIQUE
    int cfa_code;
} operator_info;

// operator table
static operator_info OP_TABLE[] = {
    { "NON",       1,  0 },
    { "ET",        2,  1 },
    { "OU",        2,  2 },
    { "IMPLIQUE",  2,  3 },
    { "PRODUIT",   2,  4 },
};
```

Ensuite, lors de la vérification sémantique, nous allons :

1. Parcourir l'AST produit par l'analyseur syntaxique.
2. Vérifier que chaque opérateur se trouve dans la table des symboles.
3. Vérifiez que l'opérateur a le nombre correct de sous-arbres enfants.

Vous pouvez constater que notre mise en œuvre est relativement simple. La vérification des types de paramètres pourrait faire l'objet d'un travail plus approfondi. Cependant, nous proposons que cette étape soit inutile puisqu'elle est déjà couverte par le processus de validation du nombre de sous-enfants.

(Les discussions sont les bienvenues ;) nous avons débattu avec nos camarades de classe et nous nous en tenons à notre idée.)

La seule ligne nécessaire dans le fichier anasem.h est la déclaration de la fonction analyze.

```
AST analyseur_semantique(AST root);
```

Pour chaque nœud de l'arbre AST, il convient d'effectuer une recherche dans la table et de vérifier s'il s'y trouve ; son nombre d'enfants est égal aux données de la table de recherche et nous procédons de manière récursive.

Nous définissons la fonction de recherche comme suit,

```
// Lookup the operator
static operator_info* lookup_operator(const char *op_name) {
    for (int i = 0; i < OP_TABLE_SIZE; i++) {
        if (strcmp(OP_TABLE[i].name, op_name) == 0) {
            return &OP_TABLE[i];
        }
    }
    return NULL;
}
```

La partie restante du code est l'itération de l'AST. Elle est mise en œuvre par deux fonctions, une fonction de point d'entrée et une fonction récursive qui s'applique à chaque nœud. Nous utilisons `switch-case` pour appliquer les différentes opérations.

```
// entry point
AST analyseur_semantique(AST root) {
    if (!root) {
        fprintf(stderr, "[Semantic Error] The AST is empty.\n");
        return NULL;
    }

    // recursive
    int result = check_semantics_recursive(root);
    if (result != 0) {
        return NULL;
    }

    return root;
}
```

La conception de la fonction récursive a été discutée ci-dessus et son implémentation se trouve dans le fichier `src/sem/anasem.c`. Nous pensons que notre implémentation est simple et facile à comprendre.

Problèmes rencontrés : nous avons longtemps discuté de la signification de cette étape. Dans notre première édition d'implémentations, cette étape n'est pas nécessaire car nous séparons la définition des opérations unaires et des opérations binaires en deux types distincts. Finalement, nous avons adopté cette solution et transformé la détection des paramètres en cette étape.

Résultat:


❯ build/test_anasem testfiles/Regles.txt
Line 1: (p1⇔p2)→((¬p1)vp2)
Valid AST.
PRODUIT
OP(IMPLIQUE)
PROP(p1)
PROP(p2)
OP(OU)
OP(NON)
PROP(p1)
PROP(p2)
Line 2: (p1v(p2∧p3))→((p1vp2)∧(p1vp3))
Valid AST.
PRODUIT
OP(OU)
PROP(p1)
OP(ET)
PROP(p2)
PROP(p3)
OP(ET)
OP(OU)
PROP(p1)
PROP(p2)
OP(OU)
PROP(p1)
PROP(p3)
Line 3: (p1vp2)→(p2vp1)
Valid AST.
PRODUIT
OP(OU)
PROP(p1)
PROP(p2)
OP(OU)
PROP(p2)
PROP(p1)
Line 4: (p1∧p2)→(p2∧p1)
Valid AST.
PRODUIT
OP(ET)
PROP(p1)
PROP(p2)
OP(ET)
PROP(p2)
PROP(p1)
Line 5: (¬(p1∧p2))→((¬p1)v(¬p2))
Valid AST.
PRODUIT
OP(NON)
OP(ET)
PROP(p1)
PROP(p2)

5. Machine Virtuelle et Compilation

5.1. Le Compilateur

Task 4 — Compilateur

On produira

1. *Un fichier* `compilateur.c` *qui contiendra*
   - *Les définitions des opérateurs* `ET`, `NON`, `OU`, `IMPLIQUE` *dans ce contexte d'évaluation par des fonctions C qui seront appelées respectivement* `Et`, `Non`, `Ou`, `Implique` *et qui seront placées dans le processeur ;*
   - *La compilation de ces opérateurs dans les tables des symboles et la machine virtuelle. Il faut faire cette étape lors de l'analyse sémantique pour que celle-ci puisse vérifier que tous les opérateurs sont bien définis et ont le bon nombre de paramètres ;*
   - *La compilation des propositions logiques à partir des arbres syntaxiques ;*
2. *Un fichier* `test_compilateur.c` *contenant une fonction* `main` *permettant de saisir une chaîne de caractères et de la compiler dans la machine virtuelle ;*
3. *Une explication des choix effectués et des problèmes rencontrés dans le rapport.*

Cette partie met en œuvre le `compilateur.c`.

La première étape consiste à définir les implémentations des opérateurs correspondants. Ces processeurs sont ensuite codés par index de manière à ce que le processeur connaisse un ensemble d'opérateurs qu'il peut exécuter.

Comme dans le contexte de l'implication logique, nous définissons le processeur comme un tableau d'opérateurs disponibles, qui est :

```
// set of instructions (operators)
void init_processor(void) {
    processeur[0] = &Non;
    processeur[1] = &Ou;
    processeur[2] = &Et;
    processeur[3] = &Implique;
}
```

La correspondance entre l'index et l'opérateur est la suivante :

```
0 -> NON
1 -> OU
2 -> ET
3 -> IMPLIQUE
4 -> PRODUIT
```

Les instructions doivent contenir les étapes suivantes :

- Extraire la valeur de la proposition T/F (Lire l'opérande)
- Exécuter
- Réécrire (Repousser sur la pile)

Selon ce principe de pipeline, nous définissons les quatre opérations comme suit :

```
// op implementations
//    read operands
// .  ex
// .  write back
void Non(void) {
    int x = Mapile.pop();
    // True = -1, False = 0
    int result = (x == -1) ? 0 : -1;
    Mapile.push(result);
}

void Ou(void) {
    int r = Mapile.pop();
    int l = Mapile.pop();
    int result = ((l == -1) || (r == -1)) ? -1 : 0;
    Mapile.push(result);
}

void Et(void) {
    int r = Mapile.pop();
    int l = Mapile.pop();
    int result = ((l == -1) && (r == -1)) ? -1 : 0;
    Mapile.push(result);
}

void Implique(void) {
    int q = Mapile.pop();
    int p = Mapile.pop();
    // p => q is (!p) or q
    if (p == -1 && q == 0)
        Mapile.push(0);   // false
    else
        Mapile.push(-1); // true
}
```

Nous avons ensuite besoin d'une fonction de mappage pour convertir le nom de l'opération (qui est une chaîne) en l'index de l'instruction correspondante.

```
// mapping
static int map_op_to_code(const char *op_name) {
    if (strcmp(op_name, "NON") == 0)       return 0;
    if (strcmp(op_name, "OU") == 0)        return 1;
    if (strcmp(op_name, "ET") == 0)        return 2;
    if (strcmp(op_name, "IMPLIQUE") == 0)  return 3;
    if (strcmp(op_name, "PRODUIT") == 0)   return 4;
    fprintf(stderr, "[Compile Error] Unknown operator: %s\n", op_name);
    return -1;
}
```

L'étape suivante consiste à transformer l'AST en VM. En définissant une VM d'un tableau pouvant contenir un maximum de 200 entiers, nous itérons récursivement chaque nœud de l'arbre, et selon le type du nœud (qu'il s'agisse d'une proposition, d'une opération ou d'une opération de produit), nous les plaçons au bon endroit, puis nous les convertissons en code.

```
// AST -> VM
extern int VM[200];
extern int vm_index;
```

Nous visitons les nœuds d'opérateurs dans l'ordre, c'est-à-dire que nous compilons d'abord la gauche, puis la droite et enfin l'opérateur lui-même. La fonction récursive est la suivante:

```
// Recursive function
static void compile_node(AST node) {
    if (!node) return;

    switch (node->type) {
    case AST_PROP: {
        int code_for_var = var_counter--;
        VM[vm_index++] = code_for_var;
        break;
    }
    case AST_OP: {
        // post-order: compile left, compile right, then operator
        compile_node(node->left);
        if (node->right) {
            compile_node(node->right);
        }
        int op_code = map_op_to_code(node->operator);
        VM[vm_index++] = op_code;
        break;
    }
    case AST_PRODUCE: {
        compile_node(node->left);
        compile_node(node->right);
        int code = map_op_to_code("PRODUIT");
        VM[vm_index++] = code;
        break;
    }
    default:
        fprintf(stderr, "[Compile Error] Unknown AST node type.\n");
        break;
    }
}
```

Enfin, nous ajoutons une fonction de point d'entrée de la fonction récursive. Nous devons enregistrer l'index vm qui commence lors du lancement du processus de compilation de cette ligne, puis nous marquons le bit de terminaison avec 99.

```
int compile_ast_to_vm(AST root) {
    if (!root) {
        fprintf(stderr, "[Compile Error] AST is null!\n");
        return -1;
    }

    // remember start address
    int start_addr = vm_index;

    compile_node(root);
```

```
    // place 99 to mark end
    VM[vm_index++] = 99;

    return start_addr;
}
```

Résultat

```
gcc -g -Wall -Wextra -o build/test_compilateur build/compilateur.o build/test_compilateur.o build/analex.o build/anasynt.o build/anasem.o build/runtime.o
gcc -g -Wall -Wextra -c src/runtime/test_runtime.c -o build/test_runtime.o
gcc -g -Wall -Wextra -o build/test_runtime build/runtime.o build/test_runtime.o build/compilateur.o build/analex.o build/anasynt.o build/anasem.o
❯ build/test_compilateur testfiles/Regles.txt
Line 1: (p1⇒p2)→((¬p1)∨p2)
Compiled to VM at start address = 0
Current VM code:
-1 -2 3 -3 0 -4 1 4 99

Line 2: (p1∨(p2∧p3))→((p1∨p2)∧(p1∨p3))
Compiled to VM at start address = 9
Current VM code:
-1 -2 3 -3 0 -4 1 4 99 -5 -6 -7 2 1 -8 -9 1 -10 -11 1 2 4 99

Line 3: (p1∨p2)→(p2∨p1)
Compiled to VM at start address = 23
Current VM code:
-1 -2 3 -3 0 -4 1 4 99 -5 -6 -7 2 1 -8 -9 1 -10 -11 1 2 4 99 -12 -13 1 -14 -15 1 4 99

Line 4: (p1∧p2)→(p2∧p1)
Compiled to VM at start address = 31
Current VM code:
-1 -2 3 -3 0 -4 1 4 99 -5 -6 -7 2 1 -8 -9 1 -10 -11 1 2 4 99 -12 -13 1 -14 -15 1 4 99 -16 -17 2 -18 -19 2 4 99

Line 5: (¬(p1∧p2))→((¬p1)∨(¬p2))
Compiled to VM at start address = 39
Current VM code:
-1 -2 3 -3 0 -4 1 4 99 -5 -6 -7 2 1 -8 -9 1 -10 -11 1 2 4 99 -12 -13 1 -14 -15 1 4 99 -16 -17 2 -18 -19 2 4 99 -20 -21 2 0 -22 0 -23 0 1 4 99

Line 6: (¬(p1∨p2))→((¬p1)∧(¬p2))
Compiled to VM at start address = 50
Current VM code:
-1 -2 3 -3 0 -4 1 4 99 -5 -6 -7 2 1 -8 -9 1 -10 -11 1 2 4 99 -12 -13 1 -14 -15 1 4 99 -16 -17 2 -18 -19 2 4 99 -20 -21 2 0 -22 0 -23 0 1 4 99 -24 -25 1 0 -26 0 -27 0 2 4 99

Line 7: (¬(¬p1))→p1
Compiled to VM at start address = 61
Current VM code:
-1 -2 3 -3 0 -4 1 4 99 -5 -6 -7 2 1 -8 -9 1 -10 -11 1 2 4 99 -12 -13 1 -14 -15 1 4 99 -16 -17 2 -18 -19 2 4 99 -20 -21 2 0 -22 0 -23 0 1 4 99 -24 -25 1 0 -26 0 -27 0 2 4 99 -28 0 0 -29 4 99
```

5.2. VM

Task 5 — VM

Proposer un modèle d'exécution permettant, en n'utilisant que la machine virtuelle, d'exécuter convenablement le code C qui suit. On produira

1. *Un fichier* `runtime.c` *définissant le code qui utilise la machine virtuelle ;*
2. *Un fichier* `test_runtime.c` *contenant une fonction* `main` *pour tester le code précédent ;*
3. *Des commentaires expliquant le fonctionnement, les choix effectués et les problèmes rencontrés pour le code proposé.*

Le fichier `runtime.c`

1. maintient une pile globale (Mapile) pour stocker les résultats intermédiaires (vrai/faux),
2. un tableau global d'instructions (VM) contenant des instructions de type bytecode et un tableau de pointeurs de fonctions (processeur[]) pour l'exécution des opérateurs.

Conformément à la question, nous fixons la valeur de l'affirmation vraie à –1 et celle de l'affirmation fausse à 0.

Nous utilisons le terme « extern » pour signifier que cet élément a été défini ailleurs et que sa valeur doit être unifiée au niveau mondial:

```
extern int VM[VM_SIZE];
extern int vm_index;

extern base processeur[50];
```

Nous définissons la pile qui lit dans la proposition les valeurs vraies et fausses comme suit:

```
#define STACK_MAX  100

typedef struct {
    int stack[STACK_MAX]; // an array
    int top;
} Stack;
```

Comme notre pile est définie en termes de file d'attente, l'opération de ‹ pop › et ‹ push › correspondante devrait être définie comme suit

```
// true = -1, false = 0
void push(int value) {
    if (s_Mapile.top >= STACK_MAX - 1) {
        fprintf(stderr, "[Runtime Error] Stack overflow.\n");
        exit(EXIT_FAILURE);
    }
    s_Mapile.stack[++s_Mapile.top] = value;
}

int pop(void) {
    if (s_Mapile.top < 0) {
        fprintf(stderr, "[Runtime Error] Stack underflow.\n");
        exit(EXIT_FAILURE);
```

```
    }
    return s_Mapile.stack[s_Mapile.top--];
}
```

Nous utilisons ici une méthode astucieuse qui nous permet de pousser et de faire sortir des éléments à la manière de « Mapile.push(-1) »

```
Mapile_t Mapile = {
    .push = push,
    .pop  = pop
};
```

La fonction `Execute(int start_addr)` lit les instructions dans le tableau VM en commençant à l'adresse spécifiée par start_addr. Lorsque l'instruction lue est inférieure à zéro, elle est considérée comme une valeur à empiler sur la pile. Si l'instruction est supérieure ou égale à zéro, elle est interprétée comme un indice d'opérateur permettant d'appeler la fonction correspondante dans le tableau processeur[]. Enfin, lorsque l'instruction vaut 99, il s'agit d'une sentinelle qui indique l'arrêt de l'exécution.

```
void Execute(int start_addr) {
    int ip = start_addr;

    while (1) {
        int instr = VM[ip++];  // Fetch next instruction

        if (instr == 99) {
            break;
        } else if (instr < 0) {
        } else {
            if (processeur[instr] == NULL) {
               fprintf(stderr, "[Runtime Error] Invalid operator index %d.\n", instr);
                exit(EXIT_FAILURE);
            }
            // Call the corresponding operator function
            processeur[instr]();
        }
    }
}
```

Le résultat:

```
❯ make
mkdir -p build
gcc -g -Wall -Wextra -c src/lex/analex.c -o build/analex.o
gcc -g -Wall -Wextra -c src/lex/test_analex.c -o build/test_analex.o
gcc -g -Wall -Wextra -o build/test_analex build/analex.o build/test_analex.o
gcc -g -Wall -Wextra -c src/syn/anasynt.c -o build/anasynt.o
gcc -g -Wall -Wextra -c src/syn/test_anasynt.c -o build/test_anasynt.o
gcc -g -Wall -Wextra -o build/test_anasynt build/anasynt.o build/test_anasynt.o build/analex.o
gcc -g -Wall -Wextra -c src/sem/anasem.c -o build/anasem.o
gcc -g -Wall -Wextra -c src/sem/test_anasem.c -o build/test_anasem.o
gcc -g -Wall -Wextra -o build/test_anasem build/anasem.o build/test_anasem.o build/anasynt.o build/analex.o
gcc -g -Wall -Wextra -c src/compil/compilateur.c -o build/compilateur.o
gcc -g -Wall -Wextra -c src/compil/test_compilateur.c -o build/test_compilateur.o
gcc -g -Wall -Wextra -c src/runtime/runtime.c -o build/runtime.o
gcc -g -Wall -Wextra -o build/test_compilateur build/compilateur.o build/test_compilateur.o build/analex.o build/anasynt.o build/anasem.o build/runtime.o
gcc -g -Wall -Wextra -c src/runtime/test_runtime.c -o build/test_runtime.o
gcc -g -Wall -Wextra -o build/test_runtime build/runtime.o build/test_runtime.o build/compilateur.o build/analex.o build/anasynt.o build/anasem.o
❯ build/test_runtime testfiles/Init.txt
Proposition evaluated to TRUE (-1)
```