GIA - ING1 - Groupe 1

F3MS - Langage de dessin

Compilateur C de fichiers `.draw` *vers Python avec Pygame*

Établi par :

CRAYSSAC Maxime, DELSUC Florian, AGUEL Fatima, DRIDI Iskander, AHMED Faïkidine, ELBAZ Benjamin et HAMMOUCHE Kévin

Groupe Projet C

Enseignante :

ZAOUCHE Djaouida

Source du projet :

Lien projet GitHub

10 janvier 2025

Département Informatique

CY TECH

Table des matières

Tableau des participants

Participant	**Rôle**	**Taches assignées**
CRAYSSAC Maxime	Chef de projet/Développeur	— Principes de compilation — Principes fondamentaux de l'analyse lexicale et syntaxique — Refonte avec liste chaînée et stockage des commandes/figures — Implémentation de la gestion des erreurs
DELSUC Florian	Développeur	— Principes fondamentaux de l'analyse lexicale et syntaxique — Développement des éléments complexes de dessin
AGUEL Fatima	Développeuse	— Conception et développement des éléments de condition complexes
DRIDI Iskander	Développeur	— Conception et développement des éléments de condition complexes
AHMED Faïkidine	Développeur	— Développement des éléments complexes de dessin
ELBAZ Benjamin	Développeur	— Développement des éléments complexes de dessin — Environnement de développement intégré (IDE)
HAMMOUCHE Kévin	Développeur	— Conception de l'implémentation globale

1 Principes de compilation et génération de code

Le processus de compilation de notre langage personnalisé repose sur un outil développé en interne qui, à partir d'un fichier source au format `.draw`, produit un script Python entièrement fonctionnel. Ce script Python, lorsqu'il est exécuté, permet de visualiser dynamiquement les formes et transformations décrites dans le fichier source.

1.1 Flux de travail global

Le flux de travail global est le suivant :

1. **Entrée utilisateur** :
   L'utilisateur fournit un fichier source, par exemple `my_draw.draw`, contenant des instructions du langage personnalisé (définition de formes, transformations, etc.).
2. **Compiler sur mesure** :
   Nous avons développé un outil de compilation, `draw_compiler`, qui intègre :
   — Analyse lexicale et syntaxique,
   — Vérification sémantique,
   — Génération de code Python.

   Cet outil est un exécutable construit à l'aide du makefile fourni.
3. **Génération de code Python** :
   Une fois le fichier `.draw` analysé, le compilateur produit un fichier Python (par exemple `draw.py`). Ce fichier contient le code nécessaire au rendu graphique (via la bibliothèque `Pygame`), reproduisant et animant les formes spécifiées dans le fichier `.draw`.
4. **Exécution dynamique** :
   L'utilisateur exécute le fichier Python généré. Un écran de visualisation s'ouvre, affichant les formes et les transformations (rotations, translations, changements de couleur, épaisseur de trait, etc.) en temps réel, selon le nombre d'images par seconde spécifié.

1.2 Fonctionnement du système de compilation

Le processus de compilation est piloté par un `makefile` dédié. Les étapes essentielles sont :

1.2.1 Gestion des sources et des dépendances

Le `makefile` définit toutes les règles nécessaires pour transformer les différentes composantes du projet (fichiers source C, fichiers Flex/Bison, etc.) en un exécutable unique, `draw_compiler`. Ce dernier est ensuite utilisé pour traduire le fichier `.draw` en `.py`.

1.2.2 Invocation du compilateur

Une simple commande `make output` (paramétrable avec des variables comme `DRAW_FILE`, `OUTPUT_PY`, ou `FRAME`) permet de lancer le processus. Le `makefile` prend en charge :

— La génération du code intermédiaire,

— L'appel à l'analyseur lexical et syntaxique,

— La compilation des différents modules,

— La création du binaire final `draw_compiler`.

Exemple : La commande suivante :

```
make output DRAW_FILE=my_draw.draw OUTPUT_PY=draw.py FRAME=5
```

Elle génère un fichier `draw.py` à partir de `my_draw.draw` et prépare un rendu à 5 images par seconde.

1.2.3 Exécutable final

L'exécutable `draw_compiler` prend en arguments :

— Le fichier source `.draw`,

— Le nom du fichier Python de sortie,

— La fréquence d'images par seconde.

Il se charge :

— D'ouvrir le fichier source,

— D'analyser le contenu (lexical et syntaxique),

— D'appeler les fonctions internes (dont `generate_python_code()`) pour produire le code Python.

À ce stade, tout est automatisé, rendant l'ensemble du processus fluide et transparent pour l'utilisateur.

1.3 Améliorations apportées au processus

Afin d'optimiser le fonctionnement, plusieurs améliorations ont été introduites :

1.3.1 Séparation claire des tâches

Les différentes étapes (analyse lexicale, syntaxique, génération de code) sont encapsulées, ce qui rend :

— Le `makefile` plus lisible,

— Le code `main.c` plus concis.

Cette séparation facilite la maintenance et l'évolution du langage.

1.3.2 Gestion des dépendances et du cache

Le `makefile` garantit que seules les parties modifiées sont recompilées :

— Si vous modifiez uniquement le fichier source du langage, la recompilation sera plus rapide.

— Les fichiers intermédiaires, tels que `lex.yy.c` ou `parser.tab.c/h`, ne sont régénérés que si nécessaire.

Cette optimisation réduit sensiblement le temps de construction global.

1.3.3 Intégration fluide du rendu dynamique

Le code Python généré inclut non seulement les commandes de dessin statiques, mais également une boucle d'événements interactive basée sur `Pygame`. Les étapes suivantes sont entièrement automatisées dans `main.c` :

— Importation et configuration du module `Pygame`,

— Initialisation de la fenêtre,

— Logique de rafraîchissement,

— Prise en compte de la variable `FRAME` (nombre d'images par seconde).

En conséquence, l'utilisateur n'a qu'à fournir un fichier `.draw` valide ; toute la chaîne de traitement, de la source au rendu dynamique, est prise en charge.

1.3.4 Extensibilité

L'architecture retenue permet d'ajouter aisément de nouvelles fonctionnalités au langage :

— Par exemple, pour introduire un nouveau type de figure (ellipse, polygone, etc.) ou une transformation (redimensionnement, cisaillement) :

  — Ajoutez les règles sémantiques correspondantes dans le code du compilateur.

  — Mettez à jour la logique de génération Python.

Cette extensibilité garantit que le système global de compilation n'a pas besoin d'être entièrement revu à chaque évolution.

2 Description de la grammaire et des règles de production

2.1 Non-terminaux

Les non-terminaux sont des éléments qui apparaissent sur le côté gauche des règles de production. Ils représentent des structures grammaticales complexes du langage. Voici la liste des non-terminaux :

- `program`
- `statement`
- `assignment`
- `function_call`
- `set_color_call`
- `set_line_width_call`
- `rectangle_call`
- `square_call`
- `circle_call`
- `draw_call`
- `rotate_call`
- `translate_call`
- `expr`
- `figure_expr`
- `line_expr`
- `point_expr`
- `rectangle_expr`
- `circle_expr`
- `square_expr`

2.2 Terminaux

Les terminaux sont les éléments de base du langage, souvent définis dans le fichier `lexer.l`. Ils incluent des tokens tels que :

- `NUMBER`
- `LPAREN`
- `RPAREN`
- `COMMA`
- `IDENTIFIER`
- `SEMICOLON`
- `EQUALS`

Ces terminaux sont utilisés pour représenter des valeurs atomiques (comme des nombres ou des mots-clés).

2.3 Règles de production

Les règles de production spécifient comment les non-terminaux peuvent être dérivés.

Programme (`program`)

```
program :
    /* Empty */
  | program statement
  ;
```

Instruction (`statement`)

```
statement :
    assignment SEMICOLON
  | function_call SEMICOLON
  | error SEMICOLON
  ;
```

Affectation (`assignment`)

```
assignment :
    IDENTIFIER EQUALS figure_expr
  ;
```

Appel de fonction (`function_call`)

```
function_call :
    set_color_call
  | set_line_width_call
  | rectangle_call
  | square_call
  | circle_call
  | draw_call
  | rotate_call
  | translate_call
  ;
```

2.4 Règles pour les appels de fonction

`set_line_width_call`

```
set_line_width_call :
    SET_LINE_WIDTH LPAREN NUMBER RPAREN
  ;
```

`set_color_call`

```
set_color_call :
    SET_COLOR LPAREN NUMBER COMMA NUMBER COMMA NUMBER RPAREN
  ;
```

rectangle_call

```
rectangle_call :
    RECTANGLE LPAREN expr COMMA NUMBER COMMA NUMBER RPAREN
  ;
```

square_call

```
square_call :
    SQUARE LPAREN expr COMMA NUMBER RPAREN
  ;
```

circle_call

```
circle_call :
    CIRCLE LPAREN expr COMMA NUMBER RPAREN
  ;
```

draw_call

```
draw_call :
    DRAW LPAREN IDENTIFIER RPAREN
  ;
```

rotate_call

```
rotate_call :
    ROTATE LPAREN IDENTIFIER COMMA NUMBER RPAREN
  ;
```

translate_call

```
translate_call :
    TRANSLATE LPAREN IDENTIFIER COMMA NUMBER COMMA NUMBER RPAREN
  ;
```

3 Analyseur lexical (Lexer)

3.1 Contexte et rôle du Lexer

Le lexer, ou analyseur lexical, est la première étape du processus d'analyse d'un langage. Son rôle est de lire le flux de caractères en entrée (par exemple, un fichier source décrivant des instructions graphiques) et de le décomposer en une suite de jetons (tokens). Ces jetons, identifiés par des catégories lexicales (mots-clés, identifiants, nombres, symboles, etc.), seront ensuite transmis à l'analyseur syntaxique (parser) afin d'être organisés selon la grammaire définie.

Dans le contexte présent, le lexer sert à interpréter une syntaxe qui semble décrire des commandes graphiques, telles que `set_color`, `set_line_width`, `point`, `line`, etc. L'objectif est donc de reconnaître ces mots-clés spécifiques, ainsi que les identifiants utilisateurs, les nombres entiers, et les signes de ponctuation utilisés dans la syntaxe (parenthèses, virgules, points-virgules, etc.). Les règles définies dans `lexer.l` contrôlent comment les chaînes de caractères sont converties en jetons, comment les erreurs lexicales sont gérées, et comment les informations de localisation (numéro de ligne) sont transmises.

3.2 Structure générale du fichier `lexer.l`

3.2.1 Section d'en-tête Flex (%{ ... %})

Entre % et %, on inclut des fichiers d'en-tête nécessaires à l'environnement global. Ici :

— **#include "../external/external.h"** :
Fichier d'en-tête externe gérant les inclusions standard (e.g., `stdio.h`, etc.).

— **#include "../common/common.h"** :
Fichier commun contenant des déclarations et fonctions utilitaires, comme `error_at_line`.

— **#include "../temp/parser.tab.h"** :
Fichier généré par Bison contenant les définitions de tokens (comme `SET_COLOR`, `POINT`, etc.) et d'autres symboles utiles.

On y trouve également :

— **`extern int yylineno;`** :
Variable fournie par Flex pour suivre le numéro de ligne.

— **`#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;`** :
Macro permettant d'actualiser la localisation du jeton à chaque action du lexer. Ceci est crucial pour le rapport d'erreurs précis.

Ces inclusions assurent que le lexer peut communiquer avec le parser (via `yylval` et les tokens) et gérer les erreurs de manière cohérente.

3.2.2 Section des options Flex

— **%option noyywrap** :
Empêche l'appel de `yywrap()` à la fin du fichier, souvent utilisé quand le traitement s'arrête une fois la fin de fichier atteinte.

— **%option nounput** et **%option noinput** :
Ces options désactivent certaines fonctionnalités obsolètes ou inutilisées de Flex, rendant le lexer plus léger.

— **%option yylineno** :
Permet de maintenir le numéro de ligne dans `yylineno`, ce qui facilite la génération de messages d'erreurs précis et l'utilisation de `YY_USER_ACTION`.

3.2.3 Section des définitions de motifs

On définit des motifs lexicaux réutilisables sous forme de macros. Par exemple :

— **DIGIT** : `[0-9]`
Un seul chiffre.

— **NUMBER** : `{DIGIT}+`
Une séquence d'un ou plusieurs chiffres, représentant un entier.

— **LETTER** : `[a-zA-Z]`
Une lettre majuscule ou minuscule.

— **IDENTIFIER** : `{LETTER}({LETTER}|{DIGIT})*`
Un identifiant commence par une lettre, suivie de zéro ou plusieurs lettres ou chiffres. Cette règle encadre la définition d'un identifiant valide.

— **WHITESPACE** : `[ \t\r\n]+`
Un ou plusieurs caractères d'espacement (espace, tabulation, retour chariot, saut de ligne).

— **COMMENT** : `#.*`
Une ligne débutant par `#` sera considérée comme un commentaire jusqu'à la fin de la ligne.

On trouve également des motifs pour les formes invalides :

— **INVALID_ID** : `{DIGIT}+{LETTER}+({LETTER}|{DIGIT})*`
Une chaîne commençant par un ou plusieurs chiffres puis une ou plusieurs lettres, etc., définissant un identifiant invalide selon les règles imposées.

— **INVALID_NUM** : `{DIGIT}*.{DIGIT}+`
Un nombre contenant un point (par exemple 3.14) qui n'est pas permis car seuls les entiers sont autorisés.

Ces définitions facilitent la maintenance et permettent d'écrire des règles plus lisibles.

3.2.4 Section des règles Flex (%% ... %%)

Ici, chaque ligne associe un motif (expression rationnelle) à une action en C. Lorsqu'un motif est reconnu dans l'entrée, l'action correspondante est exécutée. L'ordre de définition des règles compte : la première règle correspondant au texte scanné sera utilisée.

Mots-clés

— `"set_color"` `{ return SET_COLOR; }`
— `"set_line_width"` `{ return SET_LINE_WIDTH; }`
— `"point"` `{ return POINT; }`
— `"line"` `{ return LINE; }`
— `"rectangle"` `{ return RECTANGLE; }`
— `"square"` `{ return SQUARE; }`
— `"circle"` `{ return CIRCLE; }`
— `"draw"` `{ return DRAW; }`
— `"rotate"` `{ return ROTATE; }`
— `"translate"` `{ return TRANSLATE; }`

Ces règles reconnaissent des mots réservés du langage. Lorsqu'un mot-clé apparaît, le lexer renvoie un jeton correspondant qui sera interprété par le parser.

Identifiants

— `{IDENTIFIER}` `{ yylval.strval = strdup(yytext); return IDENTIFIER; }`

Si la séquence de caractères correspond au motif d'un identifiant valide, elle est copiée dans `yylval.strval`, et le token `IDENTIFIER` est retourné.

Nombres

— `{NUMBER}` `{ yylval.intval = atoi(yytext); return NUMBER; }`

Les nombres sont convertis en entier avec `atoi` et stockés dans `yylval.intval`.

Symboles de ponctuation

— `"("` `{ return LPAREN; }`
— `")"` `{ return RPAREN; }`
— `","` `{ return COMMA; }`
— `";"` `{ return SEMICOLON; }`
— `"="` `{ return EQUALS; }`

Mots-clés de contrôle et opérateurs logiques

— `"for"` `{ return FOR; }`
— `"while"` `{ return WHILE; }`
— `"if"` `{ return IF; }`
— `"else"` `{ return ELSE; }`
— `"to"` `{ return TO; }`
— `"in"` `{ return IN; }`
— `"or"` `{ return OR; }`
— `"and"` `{ return AND; }`

Espaces blancs et commentaires

— `{WHITESPACE}` `{ /* Ignore whitespace */ }`
— `{COMMENT}` `{ /* Ignore comments */ }`

Les espaces et commentaires sont ignorés, évitant tout bruit inutile pour le parser.

Règles d'erreurs

Identifiant invalide :

— `{INVALID_ID}`
`{ error_at_line(yylineno, "Invalid identifier ...", yytext); return -1; }`

Nombre invalide :

— `{INVALID_NUM}`
`{ error_at_line(yylineno, "Invalid number ...", yytext); return -1; }`

Caractère inattendu :

— `.`
`{ error_at_line(yylineno, "Unexpected character ...", yytext); return -1; }`

Chaque règle d'erreur fournit un message détaillé, guidant l'utilisateur dans la compréhension des erreurs lexicales rencontrées.

4 Analyseur syntaxique (Parser)

4.1 Contexte et rôle du Parser

Le parser a pour rôle de prendre la suite de l'analyseur lexical. Tandis que le lexer fournit une séquence de jetons (mots-clés, identifiants, nombres, symboles), le parser utilise ces jetons et les assemble en une structure syntaxique significative, selon les règles d'une grammaire formelle.

En d'autres termes, le parser vérifie que la séquence d'instructions fournie par l'utilisateur correspond bien au langage prévu, et construit des objets internes (figures, commandes) qui pourront être interprétés, exécutés, ou transformés (par exemple en code Python).

Ici, le langage semble permettre de définir des formes géométriques (points, lignes, rectangles, carrés, cercles), de les stocker dans des variables, puis de les manipuler :

— Affichage,

— Rotation,

— Translation,

— Changement de couleur,

— Épaisseur de trait, etc.

Le parser valide la syntaxe, crée des structures de données internes, et appelle des fonctions utilitaires pour stocker ces objets (ex : `add_figure`, `add_command`).

4.2 Structure générale du fichier `parser.y`

Le fichier Bison se divise en plusieurs sections standard :

4.2.1 Section d'en-tête (%{ ... %})

Entre les balises %{ ... %}, on inclut les en-têtes nécessaires au bon fonctionnement du parser :

— **#include "../external/external.h"** et **#include "../common/common.h"** :
Fournissent des déclarations communes, des fonctions d'erreur (ex : `error_at_line`), l'accès aux structures `Figure`, `Command` et la fonctionnalité d'entrées/sorties.

— **#include "../command/command.h"** :
Pour la gestion des commandes graphiques.

— **`void generate_python_code();`** :
Déclaration de la fonction qui sera utilisée ultérieurement pour générer le code Python final.

On déclare également :

— **`extern int yylineno;`** : Pour le suivi du numéro de ligne.

— L'inclusion ultérieure de **`parser.tab.h`** (généré par Bison) :
Pour récupérer `YYLTYPE` et les définitions des tokens.

Ces inclusions et déclarations préparent le terrain pour que le parser connaisse les types, fonctions et variables globales nécessaires.

4.2.2 Directives Bison

— **%error-verbose** :
Demande à Bison de produire des messages d'erreurs plus détaillés.

— **%locations** :
Active la gestion des positions (lignes, colonnes) pour les tokens et non-terminaux, permettant un meilleur retour d'information en cas d'erreur (exemple : `@1.first_line`).

4.2.3 Section %union

Cette section définit l'union C utilisée pour stocker la valeur sémantique de chaque symbole non-terminal ou token. On y trouve :

— `int intval;` :
Pour les nombres entiers.

— `char *strval;` :
Pour les identifiants (noms de variables).

— Des pointeurs vers des structures de données :
`Point *`, `Line *`, `Rectangle *`, `Square *`, `Circle *`, `Figure *`
Pour représenter les entités géométriques et les figures complexes.

Cette union permet de stocker des données hétérogènes en fonction du type de token ou non-terminal rencontré.

4.2.4 Section %token et %type

On associe chaque token défini par le lexer à un type de la %union. Par exemple :

— `%token <intval> NUMBER` :
Indique que le token `NUMBER` est associé à `intval` dans la %union.

— `%token FOR WHILE IF ELSE TO IN OR AND` :
Définit les mots-clés de contrôle de flux et d'opérateurs logiques, comme `FOR` et `WHILE`.

Des non-terminaux (via `%type`) sont également typés. Par exemple :

— `%type <pointval> point_expr` :
Cela informe Bison du type de valeur renvoyée par chaque règle.

— `%type <blockval> stmt stmt_list stmt_block loop_stmt if_stmt condition function_cal`
Indique que ces non-terminaux sont associés au type `blockval`, pour stocker des blocs de code, des instructions de contrôle, ou des appels de fonctions.

4.2.5 %start program

Définit le symbole de départ de la grammaire : `program`. C'est le point d'entrée de l'analyse syntaxique. Le parseur attend d'analyser un `program` complet.

4.3 Section des règles de grammaire (%% ... %%)

C'est le cœur du parser. On y trouve la description de la grammaire, c'est-à-dire l'ensemble des règles qui décrivent comment les jetons s'assemblent en instructions, expressions, figures, etc.

Chaque règle a la forme :

```
nom_non_terminal:
    production1 { action1 }
  | production2 { action2 }
  | ...
  ;
```

Les actions sont des blocs de code C exécutés lorsque la règle est réduite. Elles permettent de construire des structures de données, appeler des fonctions utilitaires, émettre des erreurs, etc. Les références aux valeurs sémantiques des symboles se font via `$1, $2`, etc. Les positions sont accessibles via `@1, @2`, etc.

4.3.1 program

Le programme se compose d'une liste de déclarations ou peut être vide. Chaque *statement* est traité et validé.

```
program:
    /* Empty */
  | program statement
  ;
```

La grammaire autorise un programme vide ou une suite de *statement*.

4.3.2 statement

Les *statement* peuvent être des assignations, des appels de fonction ou contenir des erreurs. En cas d'erreur, on invoque `error_at_line` avec un message complet, puis on utilise `YYABORT` pour interrompre le parsing.

Exemple de gestion d'erreur :

```
statement:
    assignment SEMICOLON
  | function_call SEMICOLON
  | error SEMICOLON {
      error_at_line(@1.first_line, "Syntax Error: Invalid statement\n\n...Explications
      YYABORT;
    }
  ;
```

Cette approche guide l'utilisateur en lui donnant non seulement l'erreur, mais aussi une liste d'exemples, de bonnes pratiques, et les erreurs courantes.

4.3.3 assignment

Gère les affectations de figures à des variables. Une règle réussie crée un `Figure *figure` à partir d'une `figure_expr` et l'ajoute via `add_figure($1, figure)`.

En cas d'erreur, un message détaillé sur la syntaxe attendue, avec des exemples valides et des erreurs courantes, est fourni.

4.3.4 function_call

Gère les appels de fonctions du langage : `set_color(...), set_line_width(...), rectangle(...),` etc. Chacune est traitée par une règle spécifique. Si un appel est invalide, un message d'erreur détaillé est fourni.

Pour chacune des fonctions du langage (ex : `set_line_width_call, set_color_call`), on récupère les arguments depuis les `$i` correspondants, on crée une structure de commande (`Command`) et on l'ajoute via `add_command(cmd)`.

Les messages d'erreur incluent :

— La forme correcte de l'appel,

— Des exemples d'appels réussis,

— Des valeurs recommandées (par exemple pour l'épaisseur de ligne),

— Les erreurs communes.

4.3.5 figure_expr

Transforme une expression représentant une figure en une structure `Figure`. Cela inclut :

— `point_expr`,

— `line_expr`,

— `rectangle_expr`,

— `square_expr`,

— `circle_expr`,

— L'utilisation d'un identifiant déjà défini comme figure.

Exemple : lorsque `figure_expr` rencontre `point_expr`, elle alloue un `Figure`, assigne le type `FIGURE_POINT` et stocke `point_expr` dans `figure->data.point`. Une gestion des échecs d'allocation mémoire est également présente, déclenchant un `error_at_line` et un `YYABORT` en cas de problème.

4.3.6 point_expr, line_expr, rectangle_expr, square_expr, circle_expr

Chaque non-terminal correspond à une construction syntaxique spécifique du langage. Par exemple :

```
point_expr:
    POINT LPAREN NUMBER COMMA NUMBER RPAREN
```

Lorsqu'un point est reconnu, il alloue un `Point` et renseigne `x` et `y`. En cas d'erreur, un message spécifique détaille :

— La syntaxe attendue,

— Des exemples de syntaxe correcte,

— Les erreurs fréquentes.

4.3.7 loop_stmt

Les *loop_stmt* gèrent les boucles `FOR` et `WHILE`. Voici des exemples d'implémentation et de gestion :

— **Boucle `WHILE`** :
Une boucle `WHILE` est traduite avec une condition. Exemple :

```
loop_stmt:
    WHILE condition '{' stmt_block '}' {
        $$ = (struct block){ .code = malloc(1024) };
        if ($$.code == NULL) {
            yyerror("Memory allocation failed in loop_stmt.");
            YYABORT;
        }
        sprintf($$.code, "while (%s) {\n%s\n}", $2.code, $4.code);
    }
```

4.3.8 if_stmt

Les *if_stmt* gèrent les structures conditionnelles, avec ou sans `ELSE`. Voici des exemples d'implémentation et de gestion :

— **`IF` seul** :
Une condition `IF` simple est traduite comme suit :

```
if_stmt:
    IF condition '{' stmt_block '}' {
        $$ = (struct block){ .code = malloc(1024) };
        if ($$.code == NULL) {
            yyerror("Memory allocation failed in if_stmt.");
            YYABORT;
        }
        sprintf($$.code, "if (%s) {\n%s\n}", $2.code, $4.code);
    }
```

— **`IF` avec `ELSE`** :
Une structure conditionnelle avec `ELSE` est traduite comme suit :

```
if_stmt:
    IF condition '{' stmt_block '}' ELSE '{' stmt_block '}' {
        $$ = (struct block){ .code = malloc(2048) };
        if ($$.code == NULL) {
            yyerror("Memory allocation failed in if_stmt.");
            YYABORT;
        }
        sprintf($$.code, "if (%s) {\n%s\n} else {\n%s\n}",
        $2.code, $4.code, $8.code);
    }
```

4.3.9 condition

Les *condition* définissent des expressions booléennes avec des opérateurs de comparaison. Voici des exemples :

— IDENTIFIER > NUMBER :

```
condition:
    IDENTIFIER '>' NUMBER {
        $$ = (struct block){ .code = malloc(64) };
        if ($$.code == NULL) {
            yyerror("Memory allocation failed in condition.");
            YYABORT;
        }
        sprintf($$.code, "%s > %d", $1, $3);
    }
```

5 Architecture C et optimisations internes

Dans le cadre de ce projet, une attention particulière a été portée à la structure et à l'organisation du code `C`, afin de garantir une maintenance aisée, une extensibilité optimale, et des performances acceptables. Notre approche repose sur plusieurs principes clés : modularité, séparation des préoccupations et utilisation de structures de données adaptées.

5.1 Séparation clairvoyante des composants

L'un des points forts de notre implémentation est le découpage logique en plusieurs modules cohérents. Chaque type de figure (*point*, *ligne*, *rectangle*, *carré*, *cercle*) est défini dans des fichiers distincts. De même :

— Les commandes (*changement de couleur*, *modification de l'épaisseur de trait*, *tracé*, *rotation*, *translation*) sont centralisées dans un module `command`.

— La liste chaînée utilisée pour stocker ces entités est gérée par un module générique `linkedList`.

Cette approche modulaire présente plusieurs avantages :

— **Lisibilité et maintenance** :
En regroupant les définitions et fonctions relatives à chaque entité dans son propre fichier :
  — Il est plus simple de naviguer dans le code,
  — Les modifications sont localisées, limitant ainsi les effets de bord.

— **Réutilisabilité** :
Le module `linkedList` peut être utilisé pour stocker n'importe quel type de données. Par exemple :
  — Figures,
  — Commandes,
  — Potentiellement d'autres éléments à l'avenir.

— **Évolutivité** :
Ajouter un nouveau type de figure ou une nouvelle commande :
  — N'implique pas de revoir tout le code existant.
  — Requiert seulement d'ajouter un nouveau fichier et d'adapter les appels nécessaires.

  La structure du code et la gestion via la liste chaînée demeurent stables.

5.2 Gestion des données avec une liste chaînée générique

Le choix d'utiliser une liste chaînée générique pour stocker figures et commandes est au cœur de notre stratégie. Cette structure, définie dans `linkedList.h`, permet :

— **Insertion Dynamique** :
On peut ajouter facilement de nouvelles entités (figures ou commandes) en début ou en fin de liste. Cela offre une grande souplesse, notamment lorsque les données sont découvertes progressivement durant l'analyse.

— **Recherche et Manipulation Simples** :
Les fonctions `find`, `contains`, ou `deleteValue` permettent de manipuler le contenu de la liste sans réécrire de code spécifique pour chaque type d'entité. Cela constitue un gain de productivité non négligeable.

— **Extension Futurisée** :
Si, à l'avenir, nous souhaitons introduire un autre type de données (par exemple, une nouvelle structure pour :

— Des groupes de figures,
— Des scènes,
— Des animations complexes),

il suffira de réutiliser la même liste chaînée. Cela réduit considérablement le risque d'introduire des bugs lors des évolutions.

5.3 Structuration des figures et des commandes

Les figures sont définies au moyen d'une énumération (`FigureType`) et d'une union, permettant de stocker divers types de figures sous une seule abstraction `Figure`. Cette union, couplée à une variable `type`, rend possible le traitement uniformisé de données hétérogènes :

— **Uniformisation des Accès** :
En connaissant le `FigureType`, le code peut accéder aux champs appropriés sans confusion. Cela simplifie la logique du code en évitant le recours à des conversions complexes ou des types distincts partout.

— **Facilité de Mises à Jour** :
Pour ajouter un nouveau type de figure (par exemple un polygone), il suffit :

— D'ajouter une entrée dans l'énumération,
— De définir une structure adaptée,
— De compléter l'union.

Le reste du code (stockage, recherche, manipulation) demeure identique.

Côté commandes, une stratégie similaire est utilisée :

— Une énumération `CommandType` (`SET_COLOR`, `SET_LINE_WIDTH`, `DRAW_POINT`, etc.),
— Une union de données associée.

Ainsi, chaque commande est traitée de manière cohérente et standardisée. Les fonctions `add_command` et la liste `command_list` centralisent la gestion des modifications à appliquer aux figures, rendant le système :

— Flexible,
— Cohérent.

5.4 Flexibilité et performance

Grâce à cette architecture, le code reste performant sans nécessiter d'optimisations prématurées. Les accès en temps linéaire ($O(n)$) de la liste chaînée sont raisonnables au vu de la nature du problème, généralement limité en taille.

Par ailleurs, la structure permet :

— De trier,
— De filtrer,
— De transformer les données ultérieurement si besoin,

simplement en ajoutant quelques fonctions utilitaires.

Avantages de l'approche

Les avantages en termes de performance résident davantage dans :

— **Clarté de la gestion des données** :
Une bonne structuration facilite l'identification des points à optimiser.

— **Adaptabilité future** :
Si les besoins en performance augmentent, il sera possible de :

  — Passer à une autre structure de données plus adaptée,

  — Introduire des caches.

5.5 Maintenabilité et facilité de débogage

La cohérence du code facilite largement le débogage. Chaque module est focalisé sur une tâche précise :

— Les figures sont définies dans `figure.[ch]`,

— Les commandes dans `command.[ch]`,

— La liste chaînée dans `linkedList.[ch]`.

En cas de bug, il est aisé de localiser le problème dans un module spécifique, sans se perdre dans une implémentation monolithique.

6 Gestion des erreurs dans le Lexer et le Parser

La gestion des erreurs est souvent l'un des éléments les plus négligés dans la conception d'un langage. Beaucoup de compilateurs ou d'interpréteurs se contentent de messages d'erreurs laconiques et peu informatifs, ce qui force l'utilisateur à perdre beaucoup de temps en conjectures et en allers-retours fastidieux.

Dans la solution proposée ici, l'objectif est radicalement différent : chaque erreur, qu'elle soit détectée au niveau lexical (lexer) ou syntaxique (parser), déclenche une réponse précise, contextualisée, et soigneusement rédigée pour aider l'utilisateur à comprendre son erreur, en saisir les règles correctes, et disposer d'exemples concrets pour réparer son code.

6.1 Philosophie générale

La démarche adoptée est pédagogique. L'idée est que l'utilisateur, qu'il soit débutant ou confirmé, doit pouvoir apprendre du message d'erreur. Au lieu de simplement signaler : `Error: invalid input`, on préfère expliquer :

- **Ce qui s'est produit** : On indique clairement la ligne et le symbole fautif.
- **Pourquoi c'est un problème** : On rappelle les règles attendues (par exemple, "Un identifiant doit commencer par une lettre, puis éventuellement contenir des chiffres").
- **Comment corriger** : On fournit une série d'exemples valides et des erreurs courantes avec explications.

Ce feedback didactique transforme chaque erreur en opportunité d'apprentissage et de progression.

6.2 Gestion des erreurs au niveau lexical

Le lexer (implémenté avec Flex) est le premier rempart contre les entrées invalides. Il transforme les caractères en jetons. Lorsqu'une entrée ne respecte pas les règles lexicales, chaque erreur a une règle dédiée dans le fichier `lexer.l`.

Exemple : INVALID_ID Si l'utilisateur tape `1circle` au lieu de `circle1`, le lexer :

- Détecte immédiatement que l'identifiant débute par un chiffre.
- Invoque `error_at_line()` pour afficher un message explicatif.

Le message inclut :

- Les règles des identifiants : doivent commencer par une lettre, ne contenir que lettres et chiffres.
- Exemples valides : `myPoint1`, `circleA`, `line123`.
- Contre-exemples : `1point` (commence par un chiffre), `my-point` (contient un tiret).

Exemple : INVALID_NUM Lorsqu'un utilisateur écrit `3.14` alors que seuls les entiers sont permis, le message :

- Explique les règles : seuls les entiers sont autorisés.
- Donne des exemples corrects : `42`, `100`, `255`.
- Met en garde contre les erreurs courantes : `3.14` (nombre à virgule), `1,000` (format invalide).

Caractères inattendus (`.`) Pour des symboles comme `{` ou `@`, le lexer :

— Liste les symboles valides (`( ) , ; =`).

— Explique leur usage (parenthèses pour les arguments, `=` pour les affectations).

— Détaille les erreurs fréquentes (ex : `{` au lieu de `(`).

6.3 Gestion des erreurs au niveau syntaxique

Le parser (implémenté avec Bison) valide la structure des instructions après le lexer. Voici des exemples d'erreurs courantes et leur gestion.

6.3.1 Affectation incorrecte

Par exemple :

```
myRect = rectangle(100,100,200,150) /* Incorrect */
myRect = rectangle(point(100,100),200,150) /* Correct */
```

Le parser fournit un message expliquant la syntaxe attendue.

6.3.2 Fonction mal formée

Exemple :

```
draw(myLine,100) /* Incorrect */
draw(myLine)     /* Correct */
```

Le message d'erreur inclut :

— **Rappel de la syntaxe** : "Usage : `draw(figure_name)`".

— **Exemples valides** : `draw(myCircle)`.

— **Erreurs fréquentes** : Utilisation de guillemets ou définition directe au lieu de variables.

6.3.3 Contrôles sémantiques

Le parser intègre des contrôles tels que :

— Vérification de l'existence d'une figure avant manipulation (ex : `rotate` sur une figure non définie).

— Vérification du type et du nombre d'arguments (ex : `line()` nécessite deux points).

En cas d'erreur, `YYABORT` interrompt l'analyse immédiatement pour éviter des messages confus en cascade.

6.4 Communication et cohérence entre Lexer et Parser

Le lexer et le parser partagent une approche cohérente :

— Le lexer bloque les fautes lexicales avant l'analyse syntaxique.

— Le parser travaille sur un flux garanti correct pour se concentrer sur la structure et la sémantique.

La fonction `error_at_line()` est utilisée pour un formatage uniforme des erreurs. La gestion des positions (via %locations et `yylineno`) permet de localiser précisément chaque problème.

6.5 Impact sur l'expérience développeur

Cette stratégie globale de gestion des erreurs améliore considérablement l'expérience utilisateur :

— **Gain de temps** : Les messages détaillés réduisent les conjectures.

— **Apprentissage continu** : Chaque erreur devient une leçon.

— **Moins de frustration** : Des messages bienveillants guident l'utilisateur.

— **Qualité du code** : Les bonnes pratiques sont adoptées plus rapidement.

7 Environnement de développement intégré (IDE)

7.1 Environnement de développement

L'IDE a été développé en C en utilisant la bibliothèque `GTK3` pour l'interface graphique et `VTE` (Virtual Terminal Emulator) pour l'intégration du terminal. Cette combinaison de bibliothèques permet de créer un environnement de développement complet et performant, offrant tous les outils nécessaires au développement.

7.2 Interface et fonctionnalités

L'interface de l'IDE est conçue pour être intuitive et efficace. La fenêtre principale est divisée en deux parties :

— Une large zone d'édition de texte.

— Un terminal intégré dans la partie inférieure.

Ce terminal joue un rôle crucial en affichant, lors de la compilation ou de l'exécution, les erreurs éventuelles, permettant aux développeurs d'identifier et de corriger rapidement les problèmes dans leur code. Le titre de la fenêtre s'adapte automatiquement pour afficher le nom du fichier en cours d'édition.

Dans l'éditeur de texte, les numéros de lignes sont affichés grâce à l'utilisation de la bibliothèque `GtkSourceView-3.0`, ce qui facilite la navigation et la lecture du code pour les développeurs.

7.3 Gestion des fichiers et édition

L'environnement offre une gestion complète des fichiers avec :

— Des fonctionnalités de sauvegarde rapide via `Ctrl+S`.

— Une option “Enregistrer sous” pour les nouveaux fichiers.

L'éditeur intègre une coloration syntaxique qui se met à jour en temps réel, notamment avec les commentaires mis en évidence en vert.

7.4 Exécution et productivité

Le système d'exécution intégré :

— Vérifie automatiquement la présence des dépendances nécessaires (notamment `Pygame`) et les installe si besoin.

— Utilise un environnement virtuel Python pour garantir une exécution isolée et stable.

Des raccourcis clavier optimisent le flux de travail :

— `Ctrl+Enter` : Compilation et exécution.

— `Ctrl+X` : Compilation et exécution directement du fichier `my_draw.draw`.

— `Échap` : Quitter l'application.