Présentation de Draw++

Réalisé par Yani A., Mathis M., Donatien L., Thomas R., Arthur N.

Dépôt GitHub : https://github.com/DonatienLinossier/Drawpp

Table des matières

Syntaxe du langage **4**
- Types 4
- Littéraux 4
- Variables 4
- Opérateurs 4
- Conditions 5
- Boucles 5
- Fonctions 5
- Curseurs 6
  - Bloc `wield` 6
  - Appel de fonction avec `wield` 6
  - Manipulation du curseur 6
- Dessin 7
- Directive `canvas` 7

IDE **8**
- Modules employés 8
- Manipulation de fichiers 8
- Le Menu 8
- Écriture du code en Draw++ 9
- Tooltip et suggestions 9
- Exécution du code 10
- Code C 11

Compilateur Draw++ **12**
- Analyseur lexical 14
  - Structure d'un *token* 14
  - Grammaire des *tokens* 15
  - Améliorations possibles 15
- Analyseur syntaxique 16
  - Arbre de syntaxe 16
  - Définitions de nœuds 17
  - Grammaire de Draw++ 18
  - Création de l'arbre à partir de *tokens* 19
  - Gestion des erreurs 20
- Analyseur sémantique 21
  - Outils 21
    - Symboles 21
    - Fonctions et variables globales 21
  - Analyse 22
    - Enregistrement des fonctions 22
    - Analyse des nœuds 22
      - `VariableDeclarationStmt` 23
      - `AssignStmt` 23
- Transpileur en code intermédiaire 24
- CTranslater 25
  - Présentation 25
  - Utilisation 25

Fonctionnement interne 30
Structure générale 30
Les subBlock 31
Les fonctions 31
Les boucles 32
Les instructions conditionnelles 33
Génération du code C 34
Axe d'amélioration 35
Compilation et *linking* 36
Module de suggestions 36
Moteur de rendu de dessin 37
Dessin des formes 37
Visionneur d'image 38

Syntaxe du langage

Le langage Draw++, fortement typé, est très inspiré du **C**, en le simplifiant à quelques endroits pour le rendre plus simple à utiliser. Il détient quelques concepts particuliers pour faciliter le dessin à l'écran.

Types

Draw++ est un langage fortement typé. Son catalogue de types est très simple :

- `float` : un réel (64-bit)
- `int` : un entier (32-bit)
- `string` : une chaîne de caractères
- `bool` : un booléen (8-bit)
- `cursor` : un curseur (voir la section « Curseurs »)

Les valeurs de type `int` sont automatiquement converties en `float` si nécessaire. La réciproque n'est pas vraie.

Littéraux

Les littéraux disponibles dans le langage, permettant d'écrire des constantes, sont :

- des nombres : entiers (`3`, `40`), décimaux (`3.47`, `3.0`)
- des chaînes : séparés par des guillemets (`"bonjour"`)
- des booléens : `true` et `false`

Variables

Les variables peuvent être définies avec ces deux syntaxes :

```
type nom = valeur_par_défaut; // Avec initialisation
type nom; // Sans initialisation
```

Par défaut, les variables ont une valeur de `0`, `false`, ou `""`. Les curseurs ne peuvent être assignés, ils sont automatiquement initialisés.

Opérateurs

Draw++ fournit toutes les **opérations arithmétiques et logiques** de base, avec ces opérateurs :

- **Opérateurs arithmétiques** : applicables sur des entiers et des réels, convertit le résultat en réel si nécessaire
  - `a + b` : addition
  - `a - b` : soustraction
  - `a * b` : multiplication
  - `a / b` : division
  - `-a` : inversion de signe
- **Opérateurs d'égalité** : applicables sur des valeurs du même type (conversion possible)
  - `a == b` : égalité
  - `a != b` : inégalité
- **Opérateurs de comparaison** : applicables sur des entiers et des réels (conversion possible)
  - `a >= b` : supérieur ou égal à

- `a > b` : strictement supérieur à
- `a < b` : strictement inférieur à
- `a <= b` : inférieur ou égal à

- **Opérateurs logiques** : applicables sur des booléens
  - `a and b` : a et b
  - `a or b` : a ou b
  - `not a` : non a

Conditions

Un bloc de code peut être exécuté conditionnellement avec le mot-clé `if`, et le mot clé `else` :

```
if ma_condition {
  // ma_condition est vraie
} else if mon_autre_condition {
  // ma_condition est fausse mais mon_autre_condition est vraie
} else {
  // Rien n'est vrai
}
```

Il est possible d'utiliser de 0 à N blocs `else if`, ainsi qu'un un bloc `else` final optionnel.

Boucles

Les boucles en Draw++ sont réalisées grâce au mot clé `while` :

```
while condition {
  // Faire des dessins tant que « condition » est vraie.
}
```

Fonctions

Le langage permet d'**appeler des fonctions prédéfinies** et de créer des **fonctions personnalisées**.

Les fonctions prennent N arguments typés, et peuvent retourner une valeur. Elles sont utilisées avec cette syntaxe : `maFonction(arg1, arg2, arg3)`.

Il est aussi possible de **fournir un curseur** lorsque l'on appelle une fonction avec le mot-clé `wield` : « `maFonction(960) wield monCurseur` » (voir la partie « Curseurs » pour plus de détails).

Il est possible de créer des fonctions avec le mot clé `fct` :

```
// Fonction avec paramètres
fct maFonction(type arg1, type arg2) {
  // Choses à faire dans la fonction
}
```

Une autre syntaxe est aussi disponible pour créer des fonctions sans paramètres :

```
// Fonction sans paramètres
fct maFonction {
```

```
  // Choses à faire dans la fonction
}
```

Les fonctions n'ont pas accès aux variables globales, et les paramètres ne peuvent être modifiés.

Curseurs

Le concept de curseur est fondamental à Draw++. Il permet de définir la **position**, **rotation** et **épaisseur** de chaque forme dessinée à l'écran.

Chaque fonction appelée utilise le **curseur actuel** pour dessiner. Au début du programme, il correspond au celui du **curseur par défaut**.

Le **curseur actuel** peut être changé de deux manières différentes.

Bloc `wield`

Le bloc `wield` permet de changer de curseur sur **une partie du programme** :

```
cursor monCurseur;
wield monCurseur {
  // Ces fonctions vont utiliser le curseur « monCurseur »

  jump(4, 5);
}
```

Ils peuvent aussi être imbriqués : le curseur du bloc parent sera à nouveau actif lorsque le bloc se termine.

Appel de fonction avec `wield`

Il est possible d'utiliser un curseur pour un **appel de fonction en particulier** en y attachant `wield` suivi du curseur :

```
cursor monCurseur;
jump(10, 10); // Déplace le curseur par défaut
jump(21, 31) wield monCurseur; // Déplace le curseur « monCurseur »
```

Manipulation du curseur

Le langage fournit quelques **fonctions prédéfinies** pour changer l'état du curseur :

- `jump(float x, float y)` : Déplace le curseur de (x, y) pixels. Le déplacement est relatif à l'ancienne position.
- `rotate(float angle)` : Tourne le curseur de l'angle donné, en radians dans le sens horaire.
- `changeThickness(float thickness)` : Change l'épaisseur du trait du curseur.
- `changeColor(int r, int g, int b, int a)` : Change la couleur du curseur en utilisant les valeurs RGBA. r, g, b et a sont des entiers allant de 0 à 255, pour le rouge, vert, bleu et l'opacité.

Dessin

Pour dessiner des formes, le langage fournit de nombreuses **fonctions prédéfinies de dessin**. Toutes les fonctions utilisent le **curseur actuel** pour dessiner, selon la **position** et la **couleur** du curseur, ainsi que d'autres attributs selon les formes :

- `circle(float r)` : Dessine un cercle creux de rayon r autour du curseur.
  Affecté par : épaisseur
- `circleFill(float r)` : Dessine un cercle plein de rayon r autour du curseur.
- `rect(float width, float height)` : Dessine un rectangle creux de largeur *width* et de hauteur *height* autour du curseur.
  Affecté par : épaisseur, rotation
- `rectFill(float width, float height)` : Dessine un rectangle plein de largeur *width* et de hauteur *height* autour du curseur.
  Affecté par : rotation
- `line(float x, float y)` : Dessine une ligne en partant du curseur jusqu'aux coordonnées (x, y).
  Affecté par : épaisseur
- `pixel()` : Dessine un pixel à la position du curseur.

Directive `canvas`

La directive `canvas <x> <y>` permet de **configurer la taille de la zone de dessin**. Les deux entiers déterminent la largeur et la longueur respectivement, et doivent être constants :

```
canvas 1280 720; // Configure une zone de dessin de taille 1280x720.
```

Cette directive doit être présente au début du programme pour être valide.

IDE

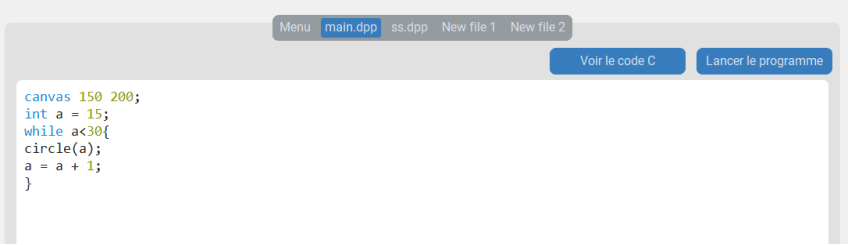
Modules employés

L'interface permet à l'utilisateur une meilleure utilisation du programme permettant de dessiner. Il a été réalisé à l'aide de customtkinter, une librairie basée sur tkinter permettant la création d'interfaces graphique reposant sur un système de composants avec des liens de parenté permettant une imbrication d'éléments de façon claire et organisée. Nous avons privilégié cette librairie plutôt que tkinter car celle-ci dispose d'une interface plus agréable.

Manipulation de fichiers

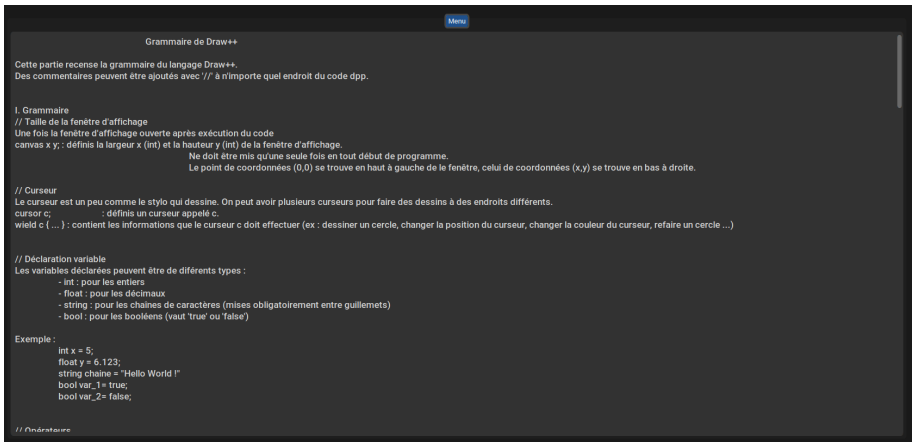
L'éditeur permet la gestion de fichiers. On peut à l'aide de boutons ou bien des raccourcis clavier habituels créer un programme avec comme nom 'New file x' où x est le nombre de fichiers créés depuis le début de la session. On peut bien sûr ouvrir un fichier déjà existant, ainsi que supprimer un fichier ouvert dans l'IDE.

Les fichiers ouverts sont accessibles sous forme d'onglets



Le Menu

L'IDE dispose d'menu sous la forme d'un onglet ouvert dès le lancement de l'interface permettant à l'utilisateur de prendre connaissance des règles de syntaxe du langage Draw++ ainsi que les fonctions existantes dans ce langage.



Écriture du code en Draw++

À l'ouverture d'un fichier et à chaque instance de modification du code, le code est analysé afin d'analyser sa syntaxe. L'ensemble des tokens est alors renvoyé permettant, en fonction de leur type, de changer le style du texte grâce à des tags appliqués au texte.

```
# Extrait du code, les tags et leur application au texte
# Configure all syntax highlighting tags
txt.tag_config("kw", foreground="#268bd2")
txt.tag_config("str", foreground="#cb4b16")
txt.tag_config("num", foreground="#859900")
txt.tag_config("cmt", foreground="#93a1a1")
txt.tag_config("err", underline=True, underlinefg="red")
tkn_list = profile("tokenize", lambda: tokenize(code_text))
for t in tkn_list:
  # First look at auxiliary text to highlight comments
  for a in t.pre_auxiliary:
      l = len(a.text)
      if a.kind == AuxiliaryKind.SINGLE_LINE_COMMENT:
          # Comment
          txt.tag_add("cmt", tidx_to_tkidx(start), tidx_to_tkidx(start + l))
      start += l

  l = len(t.text)
  if t.kind in keywords:
      # Keyword
      txt.tag_add("kw", tidx_to_tkidx(start), tidx_to_tkidx(start + l))
  elif t.kind == TokenKind.LITERAL_STRING:
      # String
      txt.tag_add("str", tidx_to_tkidx(start), tidx_to_tkidx(start + l))
  elif t.kind == TokenKind.LITERAL_NUM:
      # Number
      txt.tag_add("num", tidx_to_tkidx(start), tidx_to_tkidx(start + l))
  start += l
```

Cela permet un affichage dynamique et coloré du code comme suit :

```
canvas 250 250;
int a = 50;
float b = 20.3;
string name = "Drawing";
bool check = true;
while check and a<150{
jump(100, a);
circle(a);
a = a + 10;
}
```

Tooltip et suggestions

Si une erreur est présente dans le code, la partie erronée est soulignée en rouge. Lorsqu'on passe dessus une petite tooltip s'affiche et présente à l'utilisateur l'erreur détectée.

```
int a = "test";
```

La valeur donnée pour la variable a est du mauvais type : string donné, int attendu.

Certains cas proposent un correctif directement applicable, par exemple si le type de la variable n'est pas encore déclaré.

```
a = "test";
```

La variable a n'est pas définie.

Appliquer la suggestion : Déclarer la variable

⟹

```
string a = "test";
```

Exécution du code

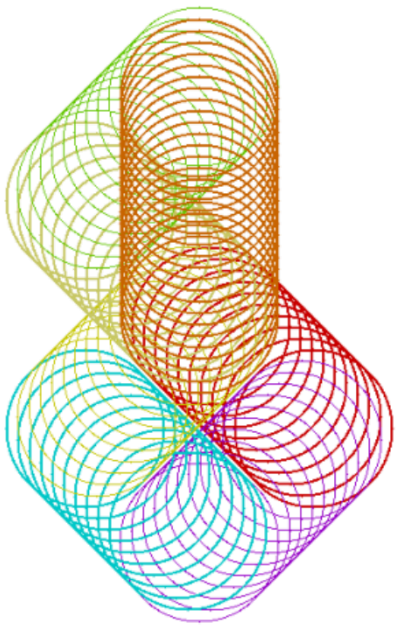
Si l'utilisateur essaye de compiler un code comportant des erreurs, les problèmes sont affichés dans le terminal permettant à l'utilisateur d'y accéder rapidement en faisant deux clics d'affiler sur la partie surlignée en bleue.

```
canvas 500 50;
int a = 5;

while a<20{
circle(a);
a=a+1
}
```

New file 1 se compile...
Error: Point-virgule manquant à la fin d'une instruction. (à [55; 55[) (double clique pour acceder)

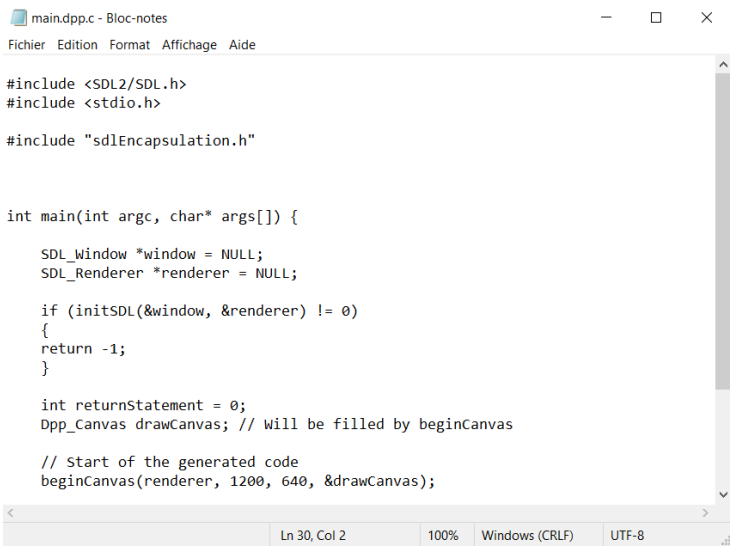
Si aucune erreur n'est détectée, le code s'exécute.

New file 1 se compile...
Succès !



Code C

L'utilisateur peut, au lieu d'exécuter le code, voir leur programme en Draw++ traduit en code C. En faisant cela, une fenêtre bloc-notes s'affichera avec leur code traduit en C.



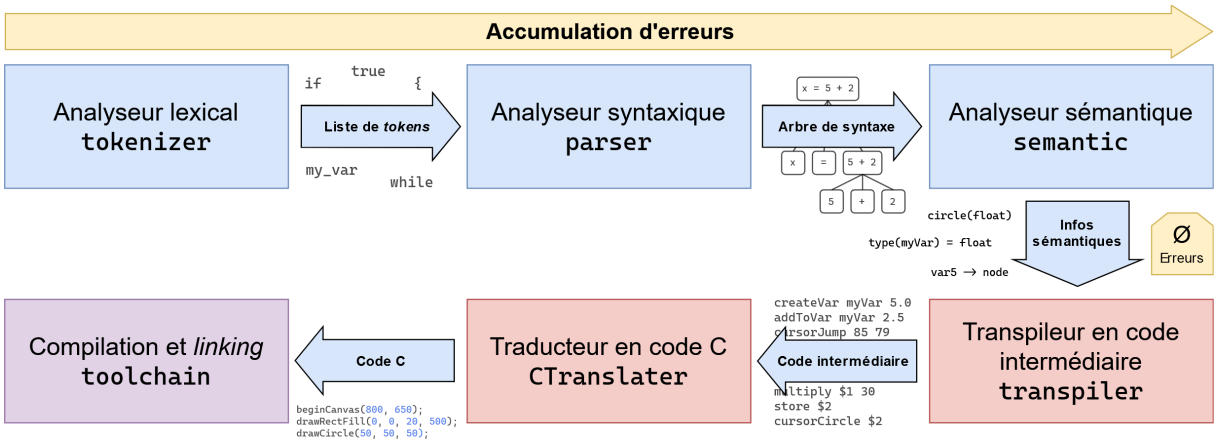
Ce code sera alors disponible dans le dossier « dpp-build ».

Compilateur Draw++

Le compilateur de Draw++ possède deux buts fondamentaux :

- transformer **du code** au langage Draw++ en **fichier exécutable** qui dessine à l'écran
- fournir à l'**IDE** une **structure facile à manier** afin de mettre en œuvre des **fonctionnalités avancées** : coloration syntaxique, erreurs en temps réel, documentation interactive, suggestions automatiques…

Le compilateur Draw++ se décompose en plusieurs parties, qui sont liées entre elles par une relation simple d'entrée/sortie :



Les **trois premières étapes** (lexical, syntaxique et sémantique ; en bleu) ont pour objectif de **traiter le code textuel brut** pour le transformer en données structurées, faciles à transformer en code C pour les étapes suivantes :

- **Analyseur lexical** (`tokenizer`) : transforme le code brut en une liste de *tokens*, des fragments de syntaxe pouvant représenter des mots-clés, des opérateurs, des littéraux, etc.
- **Analyseur syntaxique** (`parser`) : transforme la séquence de *tokens* en un arbre syntaxique, chaque nœud représentant un « bloc » du programme, rangés en deux catégories :
  - les instructions (définition/assignation de variable, déclaration de fonction, bloc « if »…)
  - les expressions (constantes littérales, opération mathématique/logique, appel de fonction…)
- **Analyseur sémantique** (`semantic`) : vérifie l'existence et la bonne utilisation des fonctions et variables utilisées, en vérifiant les types des expressions ; attache les types calculés et les nœuds référencés aux nœuds concernés (déclarations, appel à une fonction, expression de variable…)

Le code peut être erroné, comportant des erreurs de syntaxe et de sémantique. Ces **trois analyseurs** sont conçus pour être **résilients aux erreurs** contenues dans le code. Chaque étape peut traiter du code erroné, sans rencontrer de problème. Les erreurs sont « accumulées » tout le long du processus, attachées aux *tokens* et aux nœuds de l'arbre de syntaxe.

À la fin de l'analyse sémantique, si le code est erroné, la compilation ne peut continuer et s'arrête à cette étape.

Les **deux étapes suivantes** (transpileur et traducteur ; en rouge) ont pour but de **produire du code C** à partir de l'arbre de syntaxe et des informations sémantiques :

- **Transpileur en code intermédiaire** (`transpiler`) : lit l'arbre de syntaxe du programme et émet des instructions en code intermédiaire de CTranslater afin de l'exécuter
- **Traducteur en code C** (`CTranslater`) : évalue les instructions intermédiaires pour produire du code C robuste, vérifiable directement lors de la compilation

L'étape de transpilation ne peut échouer — les étapes d'analyse n'ont signalé aucune erreur — mais l'étape de traduction peut émettre des erreurs si une règle est violée (exemple : donner un rayon négatif à la fonction `circle`).

Enfin, survient l'étape de **compilation et de linking** (`toolchain`). Cette étape reçoit le code C final et le compile à l'aide du compilateur du système (`gcc` sur Linux, `cl` sur Windows), en utilisant la bibliothèque `SDL2` et notre bibliothèque `sdlEncapsulation`, fournissant les fonctions de dessin nécessaires.

Un module supplémentaire est aussi disponible : le **moteur de suggestions**. Ce petit utilitaire permet de **trouver des suggestions de correctif pour chaque erreur** avec un code d'erreur renseigné.

Analyseur lexical

L'analyseur lexical, présent dans le fichier `pydpp/compiler/tokenizer.py` du dépôt, définit un ensemble de *tokens* pour **décomposer le code de manière linéaire**, plus digeste pour l'analyse syntaxique. Ces *tokens* sont lus en utilisant une **grammaire de niveau 3** ; des expressions régulières traitées manuellement.

Structure d'un *token*

Un ***token*** est donc un « **fragment** » **textuel de code**, appartenant à l'une des quatre catégories :

Mots clés (`KW_xxx`)	Symboles (`SYM_xxx`)	Litéraux (`LITERAL_xxx`)	Spéciaux
• `fct` • `if` • `cursor` • `canvas` ⋮	• Arithmétique : `+`, `-`, `*`, `/` • Comparaison : `>`, `>=`, `<`, `<=`, `=`, `!=`, • Logique : `or`, `not`, `and` • Parenthèse : `(`, `)`, `{`, `}` • Divers : `;`, `,`, `=`	• Nombres : `75`, `29.3`, `12` • Chaînes : `"bonsoir"` • Booléens : `true`, `false`	• Identifiants : `my_var` • Fin du fichier (EOF)

Les espaces permettent de délimiter les tokens, et peuvent **contenir des erreurs** — si une chaîne n'est pas fermée, par exemple ; bien que ces erreurs soient rare en pratique.

On arrive donc à convertir du texte en *tokens*. Il serait très pratique de pouvoir faire l'inverse : **passer des *tokens* au texte**. Pour cela, il sera nécessaire **conserver les commentaires et les espaces**.

Cela se fait à l'aide du principe de **texte auxiliaire**. Il en existe trois types :
- commentaires sur une ligne (inclut le saut de ligne) : `// Mon commentaire`
- espaces (tout caractère tel que `c.isspace() == True`)
- caractères invalides, en dehors de la grammaire des tokens/commentaires : `$#`, `ff~`, etc.

Chaque *token* **enregistre le texte auxiliaire derrière lui**. Cela explique l'existence du *token* EOF, qui permet d'enregistrer le texte auxiliaire à la fin du fichier.

En résumé, la structure d'un token peut se résumer, en pseudo code :

```
class Token:
  kind: TokenKind                    # Le type du token (mot-clé, symbole, etc.)
  text: str                          # Le texte brut (auxiliaire exclus)
  value: int | float | str | bool    # La valeur constante (si c'est un littéral)
  pre_auxiliary: list[AuxiliaryText] # Les fragments auxiliaires avant ce token
  problems: list[TokenProblem]       # Les erreurs relatifs à ce token

class AuxiliaryText:
  kind: AuxiliaryTextKind            # Le type (commentaire, espace, erreur)
  text: str                          # Le texte brut de ce fragment auxiliaire
```

Prenons un morceau de code très simple pour montrer le fonctionnement de l'analyseur.

```
canvas 200 300; // Taille du canevas
circle(60); // Dessin d'un cercle de rayon 60
```

Ce code, une fois envoyé à l'analyseur, produit cette liste de *tokens*.

Type (`kind`)	Texte auxiliaire (`pre_auxiliary`)	Texte (`text`)
`KW_CANVAS`		`canvas`
`LITERAL_NUM`	`WHITESPACE`	`200`
`LITERAL_NUM`	`WHITESPACE`	`300`
`SYM_SEMICOLON`		`;`
`IDENTIFIER`	`WHITESPACE`, `SINGLE_LINE_COMMENT`	`circle`
`SYM_LPAREN`		`(`
`LITERAL_NUM`		`60`
`SYM_RPAREN`		`)`
`SYM_SEMICOLON`		`;`
`EOF`	`WHITESPACE`, `SINGLE_LINE_COMMENT`	

Grammaire des *tokens*

La plupart des *tokens*, **les mots-clés et symboles**, sont compris directement par l'algorithme de lecture en comparant les chaînes de caractères avec un algorithme optimisé.

Les **identifiants** sont composés de 1 à n caracètres dans la **catégorie Unicode « lettres » ou underscore _**.

$$\text{Identifiant} \to (\text{Lettre} \mid \_)^+$$

Les **nombres** sont composés de 1 à n chiffres (de 0 à 9, appelé Chiffre) et, optionnellement, d'une partie décimale séparée par un point.

$$\text{Nombre} \to \text{Chiffre}^+ \ \text{PartieDécimale}$$
$$\text{PartieDécimale} \to .\text{Chiffre}^+ \mid \varepsilon$$

Les **chaînes de caractères** sont des séquences de caractères quelconques délimitées par des guillemets double ("). Pour pouvoir écrire des guillemets à l'intérieurs de ces chaînes, des **séquences d'échappement** sont proposées :

- `\"` pour écrire un guillemet simple
- `\n` pour écrire une nouvelle ligne

Toute autre séquence (`\o` par exemple) produira une erreur.

Les **commentaires** sont traités après avoir lu deux « slash » d'affilée (`//`). Tous les caractères sont stockés dans un morceau de texte auxiliaire jusqu'au prochain saut de ligne, qui est **lui-même inclus dans le texte auxiliaire**.

Améliorations possibles

Bien que le *tokenizer* sans problème pour lire des fichiers, pour de gros fichiers, sa **performance laisse à désirer**. Une amélioration pertinente serait d'ajouter une ***tokenization* incrémentale**, pour **mettre à jour** uniquement les ***tokens* qui ont changé** lorsque **l'utilisateur modifie le fichier**. Un prototype a été créé, mais bien trop instable pour être utilisé avec certitude.

Analyseur syntaxique

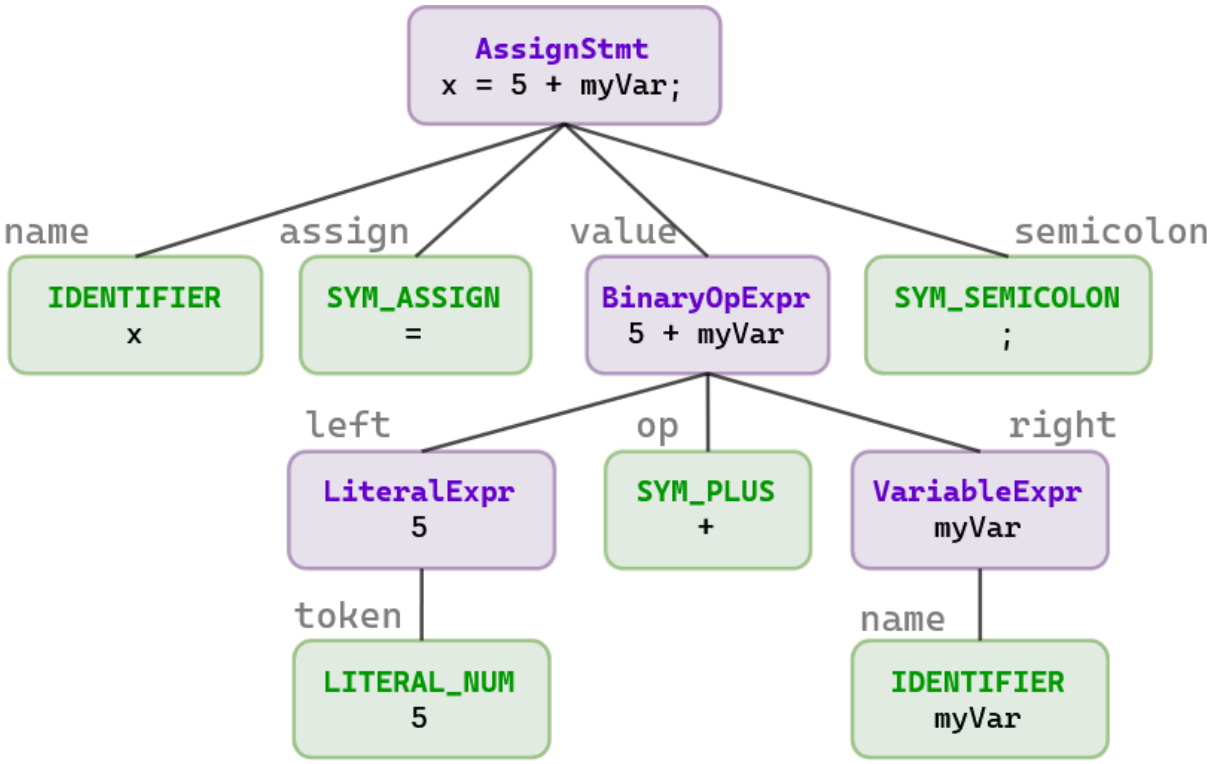
Présent dans le fichier `pydpp/compiler/parser.py`, l'analyseur syntaxique prend en entrée une liste de tokens, et fournit un arbre de syntaxe en sortie. Mais qu'est-ce un arbre de syntaxe, exactement ?

Arbre de syntaxe

Un arbre de syntaxe est, essentiellement, une **représentation structurée et hiérarchisée** du programme et de sa syntaxe. Un arbre est composé de deux types de nœuds :

- **nœuds internes** : composés de nœuds (internes ou feuilles), classifiés en trois catégories
  - **instructions** (`Stmt`) : des instructions à exécuter sur la machine, ou des déclarations/définitions
    Exemples : assignation de variable « `x = 5;` » ; déclaration de fonction « `fct f() {}` »
  - **expressions** (`Expr`) : des valeurs qui peuvent être évaluées, ayant un type (entier, booléen…)
    Exemples : un littéral « `5` » ; une opération arithmétique « `9 - 3` » ; une fonction « `f(5, 2)` »
  - **hors-format** : les nœuds spéciaux, comme le nœud racine (le programme tout entier)
- **nœuds feuilles** : contiennent un unique *token* ; un élément terminal

Commençons avec un exemple d'arbre de syntaxe. Les **nœuds internes sont en violet**, et les **nœuds feuilles sont en vert**. Le mot en gris au dessus de chaque nœud indique le nom de l'**emplacement du nœud**, par rapport à son parent — nous verrons cela en détail plus tard.



Chaque nœud peut renvoyer une **liste de ses enfants** — même les feuilles, qui donneront une liste vide —, et **connaît son nœud parent**, ainsi que son emplacement parent. Les nœuds internes ont aussi une **liste d'erreurs** qui peut être modifiée à la volée.

Cette structure en forme d'arbre permet de **restaurer le code sous format texte**, à l'aide d'un **parcours en profondeur**, en concaténant chaque nœud feuille (donc, chaque *token*). Ainsi, chaque nœud peut connaître sa **position dans le texte** (`full_span`), au format $[a; b[$, a et b étant l'index du caractère du début et de fin, respectivement.

Enfin, les nœuds sont **muables**. Il est entièrement possible d'**ajouter**, **supprimer**, ou **remplacer** des nœuds à volonté, du moment que l'emplacement accepte le nouveau nœud. Cela permet de facilement **modifier le code source**, pour mettre en place des **suggestions**.

Définitions de nœuds

Définir des nœuds manuellement — à l'aide de classes Python écrites à la main — devient vite assez pénible et chronophage. Pour accélérer le processus et rendre le code plus agréable à lire, nous avons mis en place un **générateur de classes Python**, à partir de **définitions de nœuds**.

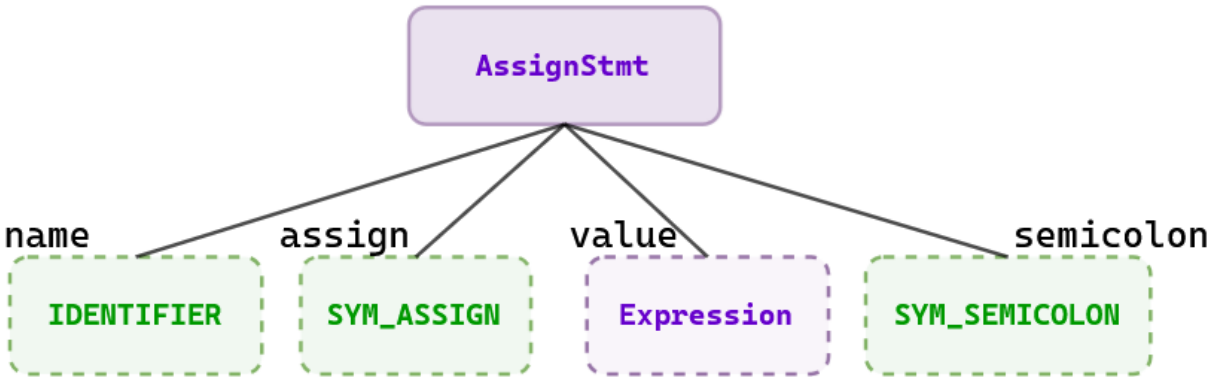
Un nœud interne est défini à l'aide d'une **liste d'emplacements**, pouvant accueillir des nœuds enfants, selon **certaines contraintes** (type du nœud, *token* particulier…). Ces emplacements possèdent un nom, et existent en deux variantes :

- **simple** (`single`) : peut contenir de 0 à 1 nœud du type désiré
- **multiple** (`multi`) : peut contenir de 0 à n nœuds du type désiré

Reprenons l'exemple précédent du nœud `AssignStmt` (`x = v;`) en étudiant sa définition en Python, à l'aide de ses emplacements définis :

```
@node_def
class AssignStmt(Statement):
    name_token = single(IDENTIFIER, doc="The name of the variable.")
    assign_token = single(SYM_ASSIGN, doc="The '=' token.")
    value = single(Expression, doc="The value to assign to the variable.")
    semi_colon = single(SYM_SEMICOLON, doc="The ';' token.")
```

Visuellement, cela ressemblera à :



On aperçoit bien les quatre emplacements du nœud d'assignation : son nom, son symbole « égal », sa valeur, et enfin le point-virgule.

Pour **générer les classes de nœud**, le générateur de classes Python (dans `pydpp/compiler/syntax/codegen.py`) **exécute** ce fichier Python contenant les **classes de prototypage**, et génère les classes correspondantes dans le fichier `generated.py`. Ces classes contiennent les fonctions nécessaires à tous les nœuds (`children`, `child_inner_nodes`) ainsi que de nombreuses fonctions et propriétés utiles (ex : un *token* `IDENTIFIER` aura une fonction `t_str()` pour avoir son texte directement).

En plus d'être pratique à écrire, ce système d'emplacements procure de nombreux avantages. Il devient bien plus simple de **modifier dynamiquement un arbre** (par exemple, pour remplacer un nœud par un autre). Il permet aussi de facilement **étudier la structure d'un nœud à l'exécution**, très pratique pour faire des fonctionnalités de **visualisation ou de débug**.

Grammaire de Draw++

Les différents types de nœuds disponibles sont issus de la **grammaire** du langage.

La grammaire de Draw++ est une **grammaire de type 2** : produisant donc un langage hors-contexte. Son **alphabet de terminaux** est composé des tous les **types de *tokens*** existants.

La grammaire est définie au **format EBNF**, disponible dans le fichier `grammar.ebnf` à la racine du dépôt. En voici son contenu (exempté de certains commentaires) :

```
(*
    GRAMMAIRE DE DRAW++
    ___________________
    - A | B  : produit par A ou B
    - [x]    : [0..1] occurrences de x
    - {x}    : [0..N] occurrences de x
    - a, b   : concaténation de a et b
    - (x)    : groupe
    - "abc"  : terminal sous forme de chaîne
*)

(* Le nœud racine : S = program *)
program = statement_list;
statement_list = {statement};

(*
   ---
   INSTRUCTIONS (STATEMENTS)
   ---
*)
statement =   variable_declaration_stmt
            | block_stmt
            | function_decl_stmt
            | if_stmt
            | while_stmt
            | function_call_stmt
            | assign_stmt
            | wield_stmt
            | canvas_stmt;

variable_declaration_stmt = type, identifier, {'=', expression}, statement_end;
function_call_stmt = function_exp, statement_end;
assign_stmt = identifier, '=', expression, statement_end;
block_stmt = '{', statement_list, '}';
function_decl_stmt = 'fct', identifier,
                     ['(', comma_sep_function_parameter_list, ')'], block_stmt;
if_stmt = 'if', expression, block_stmt,
          {'else', 'if', expression, block_stmt},
          ['else', block_stmt];
while_stmt = 'while', expression, block_stmt;
wield_stmt = 'wield', expression, block_stmt;
canvas_stmt = 'canvas', number_literal, number_literal, statement_end;
statement_end = ';';


(* -- ÉLÉMENTS INTERMÉDIAIRES POUR INSTRUCTIONS -- *)
function_parameter = type, identifier;
```

```
comma_sep_expression_list = [expression, {',', expression}];
comma_sep_function_parameter_list = [function_parameter, {',', function_parameter}];


(*
   ---
   EXPRESSIONS
   ---
   La précédence et l'associativité des opérateurs est gérée avec cette forme
   de grammaire :
       E = (E, <op>, E-1) | E-1
   Indiquant une précédence supérieure à E-1, en étant associatif à gauche.
*)
expression = or_exp;

or_exp = (or_exp, "or", and_exp) | and_exp;
and_exp = (and_exp, "and", logic_exp) | logic_exp;
logic_exp = (logic_exp, "==" | "≠" | "<" | "≤" | ">" | "≥", additive_exp)
            | additive_exp;
additive_exp = (additive_exp, "+"|"-", multiplicative_exp) | multiplicative_exp;
multiplicative_exp = (multiplicative_exp, "*"|"/", unary_exp) | unary_exp;
unary_exp = (["-"|"not"], unary_exp) | function_exp;
function_exp = (identifier,
                "(", comma_sep_expression_list, ")",
                ["wield", expression]) | parens_exp;
parens_exp = ("(", expression, ")") | value;
value = literal | identifier;

(*
    ---
    LITÉRAUX, TYPES & IDENTIFIANTS
    ---
*)
literal = number_literal | bool_literal | string_literal;

(*
  Ces éléments sont lus de manière spéciale par l'analyseur lexical,
  voir sa partie consacrée pour en connaître les détails.
*)
number_literal = ? élément terminal défini par l'analyseur lexical ?;
string_literal = ? élément terminal défini par l'analyseur lexical ?;
bool_literal = "true" | "false";

type = "int" | "float" | "bool" | "string" | "cursor";
identifier = ? élément terminal défini par l'analyseur lexical ?;
```

Création de l'arbre à partir de *tokens*

Avec une liste de *tokens*, l'analyseur syntaxique **produit un nœud racine**, le nœud `Program`. Ce nœud contient simplement **une liste d'instructions** (`Stmt`) à exécuter.

Pour lire ces instructions — et quelques expressions —, l'analyseur utilise une technique à base de **fonctions récursives** (*recursive descent*), très adapté aux langages simples de type 2, elle permet de facilement **détecter et signaler des erreurs** lorsqu'un morceau de code est erroné.

Pour lire des expressions d'opérations binaires (addition, multiplication, « OU » logique, etc.), une autre technique est utilisée : le ***precedence climbing***, issu de Martin Richards et Colin Whitby-

Strevens. Cette technique permet de conserver une **précédence correcte** — c'est-à-dire, `1+2/3` sera `1+(2/3)` et non pas `(1+2)/3` — en lisant séquentiellement les opérateurs, puis en « **reportant** » **les opérateurs de plus haute précédence** à un autre appel de la fonction.

Gestion des erreurs

Le code erroné étant monnaie courante lorsque l'on écrit du code, il est important pour l'analyseur syntaxique de **continuer à reconnaître le plus de syntaxe** possible. Il est aussi important de fournir un **message d'erreur les plus clair possible**. Pour ce faire, l'analyseur utilise **deux stratégies** pour continuer à lire le code lorsqu'une erreur survient.

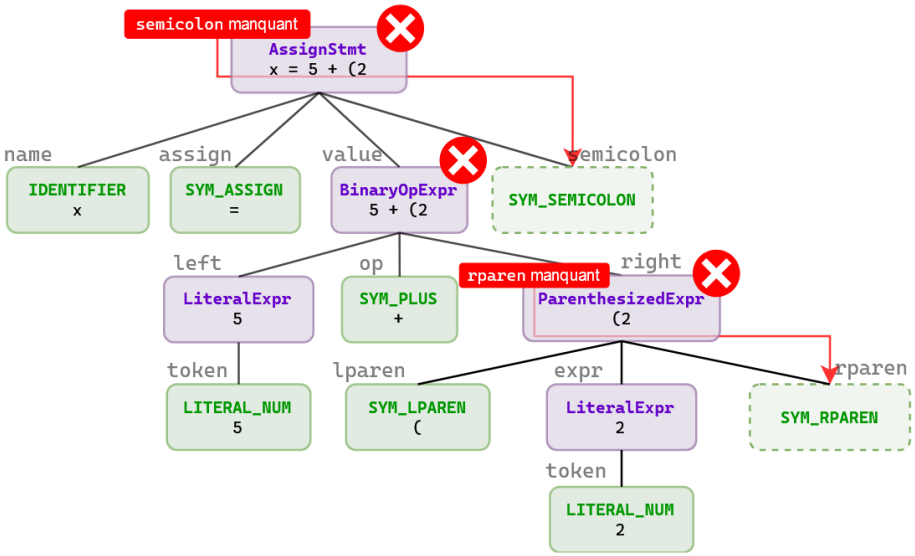
Le plus souvent, un **élément attendu** n'est simplement **pas présent** : un point-virgule manquant, une parenthèse manquante… Dans ce cas, l'analyseur continue normalement en **supposant que cet élément existe**, et ajoute une erreur au nœud concerné. Une trentaine d'erreurs sont signalées en utilisant cette technique.

Pour plus de précision, la lecture d'une liste d'argument (`f(1, 2, 3)`) est doté d'un **algorithme de lecture particulier**. Au lieu d'abandonner la lecture lorsqu'un élément n'est pas une expression (`f(1, else)`), ce dernier va **continuer de lire des *tokens*** jusqu'à la prochaine virgule, parenthèse, accolade, ou point-virgule. Ainsi, même une expression aussi bancale que `f(while + if, wield)` sera comprise par l'analyseur.

Pour stocker les erreurs, chaque **nœud interne** possède une **liste de problèmes** associée. Les problèmes enregistrent notamment **l'emplacement concerné par l'erreur**, ce qui est très utile lorsqu'un élément est manquant. En pseudo-code, un problème peut être défini comme tel :

```
class InnerNodeProblem:
  message: str               # La description du problème
  severity: ProblemSeverity # L'importance du problème (erreur, avertissement, info)
  code: ProblemCode          # Le code d'erreur associé
  slot: NodeSlot | None      # L'emplacement concerné par le problème.
                             # Pas d'emplacement → le nœud entier est concerné
```

De plus, chaque nœud possède un attribut `has_problem` pour savoir s'**il est erroné, ou ayant au moins un descendant erroné**. Cela permet de facilement **parcourir tous les nœuds erronés** et récupérer les erreurs facilement. (Montré ci-dessous par une croix sur un cercle rouge.)



Analyseur sémantique

L'analyse sémantique est réalisée à l'aide de la fonction `analyse`, présente dans le fichier `pydpp/compiler/semantic.py`, et qui prend en paramètre un arbre syntaxique. Son intérêt est de vérifier la cohérence et la validité du code en parcourant tous les nœuds de l'arbre.

Outils

Symboles

Les **symboles** sont des objets qui contiennent des informations sémantiques sur des entités du programme comme les variables, fonctions, paramètres ou expressions. Ces symboles stockent généralement le nom, le type et le nœud sans toutefois s'y limiter (ex : les paramètres dans le cas d'une fonction) et ils associent des informations sémantiques nœud de l'arbre (comme des assignations d'erreur).

Il existe des symboles pour :

- les variables : `VariableSym`
- les fonctions : `FunctionSym`
- les paramètres de fonction : `ParameterSym`
- les expressions (`Expr`) : `ExpressionSym`
- les instructions (`Stmt`) : `AssignSym`

Exemple de la classe `VariableSym` :

```
class VariableSym:
    def __init__(self, name: str, type: SemanticType, node: VariableDeclarationStmt
| None, builtin_val=None):
        self.name = name
        self.type = type
        self.node = node
        self.built_in_val = builtin_val

    def __str__(self):
        return f"VariableSym({self.name}, {self.type})"
```

Fonctions et variables globales

Pour permettre l'analyse sémantique, les fonctions et variables globales doivent pouvoir être stockées et accessibles. C'est le rôle de `builtin_funcs` et `builtin_vars`, deux dictionnaires contenant dès l'origine les fonctions et variables globales natives à Draw++.

```
builtin_funcs = {
    "circle": FunctionSym(
        name="circle",
        return_type=SemanticType.NOTHING,
        parameters=[ParameterSym("r", SemanticType.FLOAT, None),],
        c_func_name="cursorDrawCircle",
        node=None,
        doc="Dessine un cercle creux de rayon r autour du curseur."
    ),
    ...
}
builtin_vars = {
```

```
    "PI": VariableSym("PI", SemanticType.FLOAT, None, 3.14159265358979323846),
}
```

A noter : dans la fonction `analyse`, ces dictionnaires possèdent les noms `global_funcs` et `global_vars`. Les fonctions et variables globales déclarées dans le code seront donc ajoutées dans ces dictionnaires au fur et à mesure de l'analyse des nœuds.

Analyse

L'analyse sémantique se fait dans la fonction `analyse` (contenant elle-même des sous-fonctions). Dans un premier temps, la fonction analyse tout l'arbre passé en paramètre pour enregistrer les **fonctions** du programme. Cette étape est nécessaire pour l'analyse des autres nœuds de l'arbre car ces derniers peuvent appeler les fonctions créées. Ensuite, la fonction analyse tous les **nœuds** à l'aide de la fonction `analyse_node`.

Enregistrement des fonctions

L'enregistrement des fonctions est possible grâce à la fonction `register_function` qui vérifie :
- la validité du nom de la fonction
- que le nom de fonction n'est pas déjà utilisé par une autre fonction
- la validité des noms et des types de chaque paramètre
- que plusieurs paramètres n'ont pas le même nom

Si l'analyse s'est déroulée sans encombres, la fonction est ajoutée au dictionnaire des fonctions globales `global_funcs` et à `function_to_sym`. En cas de nouvelle déclaration de fonction, ce dictionnaire permet de savoir si une fonction de même nom a été déclarée.

À noter : Le dictionnaire `global_funcs` contient toutes les fonctions globales et est initialisé avec des fonctions natives.

Analyse des nœuds

Une fois les fonctions enregistrées, l'arbre est à nouveau analysé nœud par nœud, chaque nœud ayant cette fois un traitement sémantique qui lui est propre. La fonction `analyse_node` est chargée de cela. Elle prend comme paramètre un nœud (qui est une instruction) et a pour but d'identifier sa classe pour lui appliquer une analyse spécifique.

À noter : la fonction `analyse_node` est récursive et analyse les nœuds enfants du nœud fourni. Par exemple : si le nœud d'une fonction est analysé, alors les nœuds enfants (c'est-à-dire le contenu de la fonction) le seront aussi.

La classe du nœud peut être :
- `VariableDeclarationStmt` : déclaration de variable
- `FunctionDeclarationStmt` : déclaration de fonction
- `IfStmt`, `WhileStmt` ou `ElseStmt` : if, else ou while
- `AssignStmt` : assignation de variable
- `WieldStmt` : curseur
- `CanvasStmt` : dimensions de fenêtre d'affichage
- `Expression` : une expression

Nous allons par la suite étudier les 2 exemples de l'utilisation de la fonction `analyse_node` dans les cas de l'analyse d'une `VariableDeclarationStmt` et d'une `AssignStmt`. Les fonctions ont quant à elle

déjà été analysées juste avant et pour le cas de `IfStmt`/`WhileStmt`/`ElseStmt` il suffit de vérifier que la condition est un booléen.

`VariableDeclarationStmt`

Pour procéder à l'analyse de la déclaration de la variable, la fonction `register_variable` est appelée. Cette dernière est chargée de :

- détecter si le nom de variable est déjà utilisé
- détecter si la variable est globale ou locale (à une fonction)
- vérifier si une variable a le même nom
  - variable globale : on vérifie avec les noms des variables dans `global_vars`
  - variable locale : on vérifie si un argument (de la fonction) a le même nom
- appeler la fonction `register_expression` : cette fonction va regarder si la variable déclarée et sa valeur sont de même type.
- enregistrement de la variable dans `variable_to_sym` (utilisée pour les variables locales et globales) `global_vars` (uniquement pour les variables globales)

`AssignStmt`

Pour procéder à l'analyse de l'assignation de valeur à une variable existante, `analyse_node` effectue les vérifications suivantes :

- vérification que la variable est définie
- vérification que la variable est un paramètre, si c'est le cas on ne peut pas assigner de valeur à un paramètre (erreur)
- vérification qu'on assigne pas une valeur à une constante (`VariableSym.built_in_val`) ce qui est impossible
- vérification qu'on assigne une valeur du bon type
- enregistrement valeur (fonction `register_expression`)

A la fin de l'analyse sémantique, la fonction `analyse` retourne un objet de classe **`ProgramSemanticInfo`** contenant toutes les informations relatives à l'analyse sémantique à savoir :

- toutes les variables et fonctions globales
- les informations sémantiques sur tous les nœuds (dictionnaire « `_to_sym` »)

```
def __init__(self,
         global_functions: dict[str, FunctionSym],
         global_variables: dict[str, VariableSym],
         function_to_sym: dict[FunctionDeclarationStmt, FunctionSym],
         variable_to_sym: dict[VariableDeclarationStmt, VariableSym],
         expr_to_sym: dict[Expression, ExpressionSym],
         assign_to_sym: dict[AssignStmt, AssignSym],
         canvas_width: int | None,
         canvas_height: int | None):
self.global_functions = global_functions
self.global_variables = global_variables
self.function_to_sym = function_to_sym
self.variable_to_sym = variable_to_sym
self.expr_to_sym = expr_to_sym
self.assign_to_sym = assign_to_sym
self.canvas_width = canvas_width
self.canvas_height = canvas_height
```

Transpileur en code intermédiaire

Le **transpileur** (présent dans `pydpp/compiler/transpiler.py`) permet de transformer l'**arbre de syntaxe annoté** en **instructions intermédiaires de CTranslater**.

L'algorithme parcourt tous les nœuds, avec une recherche en profondeur, et **émet les instructions** nécessaires.

Chaque instruction simple (déclaration de variable, appel de fonction) est transformée en instruction simple (`createVar`, `myFunc`).

Les expressions sont évaluées à l'aide de **variables intermédiaires** créée pour chaque nœud. Ces variables sont au format `${i}`, avec `i` un entier qui s'incrémente à chaque nouvelle variable intermédiaire créée.

Les instructions en bloc (déclaration de fonction, bloc « if », boucles « while ») créent des **instructions de bloc**, avec des sous-blocs qui sont créés pour chaque embranchement (`if`, `else`, etc.)

Le transpileur conserve une **« pile » de curseurs**, ayant toujours en premier élément le curseur par défaut (appelé `!default_cursor`). La pile est modifiée lorsque l'on utilise des blocs `wield` (permettant d'utiliser un curseur sur un bloc entier d'instructions), changeant le curseur qui est **envoyé aux fonctions**.

Chaque fonction définie par l'utilisateur est, en interne, agrémentée d'un **paramètre supplémentaire** : **le curseur**. Lorsque l'on appelle une fonction, le transpileur **récupère le curseur de la pile** pour l'envoyer à la fonction. Ce curseur peut être remplacé en utilisant le mot clé `wield` après l'appel de fonction (`circle(5) wield c;`).

CTranslater

CTranslater est un module Python conçu pour la génération automatique de code en langage C. Il fournit des outils permettant de créer un fichier C compilable à partir d'une suite d'appels de méthodes Python. Dans notre projet, c'est le transpiler qui fait appel aux méthodes fournit par le CTranslater. L'objectif principal de CTranslater est de simplifier la génération de code graphique en C à l'aide de la bibliothèque SDL. Ainsi, les utilisateurs peuvent produire des fichiers permettant d'afficher des dessins ou des animations définis à partir des instructions fournies.

Présentation

CTranslater fonctionne en deux étapes principales. La première, appelée étape de remplissage, consiste à insérer des instructions dans le module via les méthodes proposées. Cela correspond à la phase de conception, où les actions à réaliser sont définies. Ensuite vient l'étape d'exécution et de génération, où le module exécute les instructions pour produire un fichier C compilable. Ce fichier intègre également le code nécessaire à l'utilisation de la bibliothèque SDL pour afficher les dessins ou animations spécifiées.

CTranslater propose une large palette d'instructions réparties en plusieurs catégories. Il permet d'effectuer des opérations arithmétiques (comme l'addition, la soustraction ou la division) et des comparaisons standards (par exemple, égalité ou inégalités). En termes de capacités graphiques, CTranslater inclut des instructions pour manipuler un curseur, qui se déplace sur une zone graphique et permet de tracer des lignes et ainsi que des formes tel que des carrés ou cercles.

Des fonctionnalités avancées sont également disponibles, notamment les instructions conditionnelles pour introduire des branches logiques, les boucles pour répéter des actions ou des dessins, ainsi que la possibilité de créer des fonctions pour structurer et organiser le programme généré.

Utilisation

Instanciation

La première étape consiste à créer une instance de l'objet CTranslater. Cela se fait en important le module et en initialisant l'objet avec le nom du fichier où le code généré sera enregistré :

```
import pydpp.compiler

myCTranslater = CTranslater("filename")
```

Ici, « filename » désigne le fichier dans lequel le code généré sera sauvegardé.

Construction

Cette étape consiste à définir **les instructions** qui seront exécutées par CTranslater. Ces instructions déterminent les actions ou les dessins souhaités. L'ajout d'une instructions se fait via la méthode :

```
myCTranslater.add_instruction("drawCircle", 10, 15, 5)
```

Le premier argument est le nom de l'instruction souhaitée, et les autres sont les paramètres attendus par cette instruction. Ainsi ici, on vient d'ajouter une instruction qui dessinera un cercle aux coordonnées (10, 15) de rayon 5.

CTranslater permet aussi l'utilisation de **variable**. Les types numérique(int et float), les strings et les booléens sont supportés. Pour les utiliser, il suffit de les initialiser avec :

```
myCTranslater.add_instruction("createVar", "variableName", 5)
```

Il est ensuite possible de les utiliser directement en encadrant leur nom avec « varCall() ».

Ainsi en reprenant l'exemple du cercle ci-dessus nous aurions :

```
myCTranslater.add_instruction("createVar", "x", 10)
myCTranslater.add_instruction("createVar", "y", 15)
myCTranslater.add_instruction("createVar", "radius", 5)

myCTranslater.add_instruction("drawCircle", varCall("x"),
                              varCall("y"), varCall("radius"))
```

Bien évidemment, il est possible d'effectuer des opérations sur ces variables. Il est notamment possible de leur affecter une nouvelle valeur, de les additionner, les soustraire, les multiplier et autre. Il est aussi possible d'effectuer des opérations de comparaison tel que « supérieur à », « plus petit que » ou « égal à ».

CTranslater met aussi à disposition des instructions plus complexes comme **les boucles** ou **les instructions conditionnelles**. Leur création se fait en plusieurs parties. Pour commencer il faut déclarer l'utilisation d'un de ces blocs :

```
#Creation d'un bloc while
myCTranslater.createWhileLoop()
```

```
#Creation d'un bloc if else
myCTranslater.createConditionalInstr()
```

Le remplissage de ces instructions se fait par bloc. Pour les boucles, le premier bloc représente la condition pour que la boucle s'exécute et le second le corps de la boucle. Pour les instructions conditionnelles, le premier bloc représente la condition, le second le corps du bloc if, et le troisième le corps du bloc else. Pour finir un bloc et passer au prochain, il suffit d'écrire :

```
myCTranslater.endBlock();
```

Le CTranslater placera les nouvelles instructions directement dans le bloc suivant. Dans le cas ou il s'agissait du dernier bloc de l'instruction, les prochaines instructions seront placés dans l'élément parent.

Par exemple, pour créer une boucle qui sera exécutée qu'une seule fois, on peut faire :

```
import pydpp.compiler
```

```
#Initialisation
myCTranslater = CTranslater("filename")


#Construction
myCTranslater.add_instruction("createVar", "myvar", True)

myCTranslater.createWhileLoop()

myCTranslater.add_instruction("functReturnStatement", VarCall("myvar"))

myCTranslater.endBlock()

myCTranslater.add_instruction("createVar", "myvar", False)

myCTranslater.endBlock()
```

A noter ici que dans le premier bloc il s'agit d'une instruction functReturnStatement. En effet, pour évaluer une condition, que ce soit pour les boucles ou les instructions conditionnelles, c'est la valeur retourné qui est prise en compte. Il est ainsi possible d'effectuer une multitude d'opération dans la condition et seule la valeur retournée sera évaluée.

CTranslater permet aussi la création et l'utilisation de **fonction**. Leur fonctionnement reprends celui autres instructions complexes avec une première étape de définition, à la différence que leur définition prends des arguments. Le premier argument est le nom de la fonction créée, et le second une liste de paramètre pour la fonction. Ainsi, au lancement de la fonction il faudra spécifier le même nombre de d'argument, et ceux-ci seront accessible comme des variables au nom des parametres.

```
myCTranslater.createFunc("TestFunction", ["bool", "i"])
```

Comme pour les autres instructions complexes il faut maintenant remplir cette fonction avec :

```
myCTranslater.add_instruction("instructionName", "parameters")
```

Il est aussi possible de faire retourner une valeur par la fonction. Pour cela il suffit d'utiliser la fonction :

```
myCTranslater.add_instruction("functReturnStatement", "the_returned_value")
```

Une fois la fonction définit, pour finir l'étape de création il suffit d'ajouter :

```
myCTranslater.endBlock()
```

La fonction devient ainsi appelable comme toute autre instruction avec :

```
myCTranslater.add_instruction("TestFunction", True, 5)
```

Enfin pour conclure, il est à noter que toutes les instructions complexes sont imbricables. Il est ainsi totalement faisable de faire de cumuler les boucles, instructions conditionnelles et fonctions en parallèle, mais aussi les uns à l'intérieur des autres.

Compilation

Une fois la phase de construction finies, il ne reste plus qu'a lancer la compilation avec :

```
myCTranslater.run()
```

Après cette étape, le code C correspondant est ainsi crée dans la fichier « filename » spécifié à l'étape d'initialisation.

Afin d'imager une utilisation complète du module, voici un programme avec une boucle, une instruction conditionnelle et une fonction.

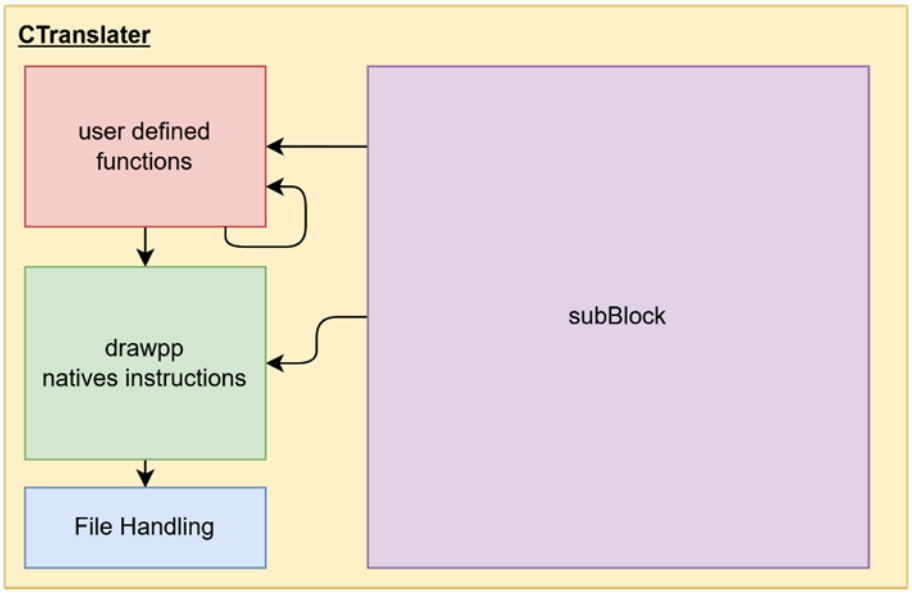
```
import pydpp.compiler

#Initialisation
myCTranslater = CTranslater("monFichier.c")

#Creation
myCTranslater.add_instruction("createVar", "bool", True)
myCTranslater.add_instruction("createVar", "cmpt", 1)

#def TestFunction(bool, i) {
myCTranslater.createFunc("TestFunction", ["bool", "i"])

#while (
myCTranslater.createWhileLoop()
myCTranslater.add_instruction("functReturnStatement", VarCall("bool"))
myCTranslater.endBlock()
# ) //end while condition
# {


test.add_instruction("addToVar", "i", 1)


#if (
myCTranslater.createConditionalInstr()
myCTranslater.add_instruction("functReturnStatement", VarCall("bool"))
myCTranslater.endBlock()
#) { // end if condition

myCTranslater.add_instruction("createVar", "bool", False)
myCTranslater.endBlock()
# } else {

myCTranslater.endBlock()
# } // end else


myCTranslater.add_instruction("drawCircle", VarCall("i"), VarCall("i"),
VarCall("i"))
```

```
myCTranslater.endBlock()
# } //end While

myCTranslater.endBlock()
# } //end TestFunction

myCTranslater.add_instruction("TestFunction", VarCall("bool"), VarCall("cmpt"))

myCTranslater.add_instruction("drawCircle", VarCall("cmpt"), VarCall("cmpt"),
VarCall("cmpt"))


#Execution
myCTranslater.run()
```

Fonctionnement interne

Structure générale

Le fonctionnement de CTranslater peut être schématisé de la façon suivante :



Schématisation du fonctionnement de CTranslater

1. Le SubBlock

- Contient l'ensemble des instructions du code, telles que définies par l'utilisateur.
- Agit comme un conteneur où les instructions sont organisées avant d'être traduites en code C.
- Ces instructions peuvent être des appels directs aux fonctions natives de drawpp ou aux fonctions définies par l'utilisateur.

2. Le bloc des fonctions utilisateur

- Stocke les fonctions créées par l'utilisateur.
- Ces fonctions peuvent être appelées depuis le SubBlock ou d'autres fonctions utilisateur.
- Favorise la réutilisabilité et la modularité dans le code utilisateur.

3. Le bloc des instructions natives

- Contient des instructions pré-définies, y compris celles dédiées aux graphiques (via SDL).
- Propose des primitives pour les opérations de dessin, les calculs, la logique.
- Ces instructions sont prêtes à l'emploi et servent de base pour l'exécution des subBlocks.

4. Le Block de gestion des fichiers

- Gère les aspects liés à la génération, l'écriture, et la sauvegarde du fichier C final.
- S'occupe de l'intégration des en-têtes nécessaires et des appels aux bibliothèques, comme SDL.
- Garantit que le fichier généré est correctement structuré et compilable.

Les subBlock

Un SubBlock est une unité de base dans la structure de CTranslater. Il gère les instructions et les données nécessaires pour exécuter une série d'actions spécifiées par l'utilisateur.

Un subBlock est constitué de 2 parties principales :

- La partie instruction, qui stocke les instructions rentrées par l'utilisateur.
- La partie mémoire, qui stocke les variables lors de l'exécution des instructions. A noter qu'un bloc parameters existe dans la partie mémoire. Celui-ci représente des données qui peuvent être transmise au subBlock avant son exécution. Cette fonctionnalité est notamment utile quand le subBlock partage la même mémoire que l'élément parent, ou qu'il s'agit d'une fonction pouvant prendre des arguments.

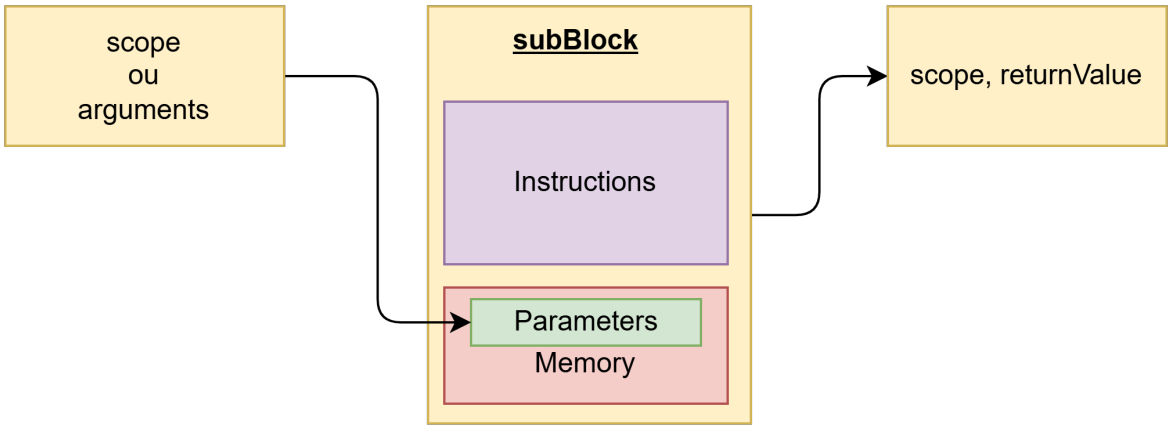


Schéma de la structure d'un subBlock

Lors de son éxécution, un subBlock peut prendre en entrée une liste de variables, qui seront ajoutés à la mémoire (bloc parameters). En sortie d'éxécution, un subBlock retourne une pair (liste de variable, returnValue). La liste de variable peut être utilisé pour appliquer aux parents les modifications fait au sein du bloc, notamment dans le cas d'un scope partagé. La returnValue réprésente, comme son nom l'indique, la valeur retourné par le subBlock, qui sera géré par l'élément parent.

Les instructions complexes, telles que les boucles, les fonctions et les instructions conditionnelles, reposent sur le subBlock, qui constitue la base de leur fonctionnement. Chaque instruction complexe utilise le subBlock pour gérer ses mécanismes internes, notamment en exploitant les entrées pour transmettre des données et les sorties pour retourner des résultats ou des états.

A noter que c'est le fait que les instructions complexes soient composés de subBlock (subBlock supportant donc eux même les instructions complexes) qui permet d'imbriquer sans limite les instructions complexes.

Les fonctions

A la création d'une fonction, l'utilisateur doit spécifier une liste de paramètre. Cette liste peut être vide dans le cas d'une fonction sans paramètre. Ainsi, lors de l'appel d'une fonction les arguments sont vérifiés et reliés aux paramètres.

Une fois liés à leurs paramètres, les arguments sont envoyés au subBlock. Ainsi les arguments sont utilisables directement dans le subBlock.

Par exemple :

Soit la fonction « test(int a, float b) ». Si on exécute la fonction avec « test(42, 52.2) », les variables a et b seront accessibles à l'intérieur du subBlock avec pour valeurs respectives 42 et 52.2.

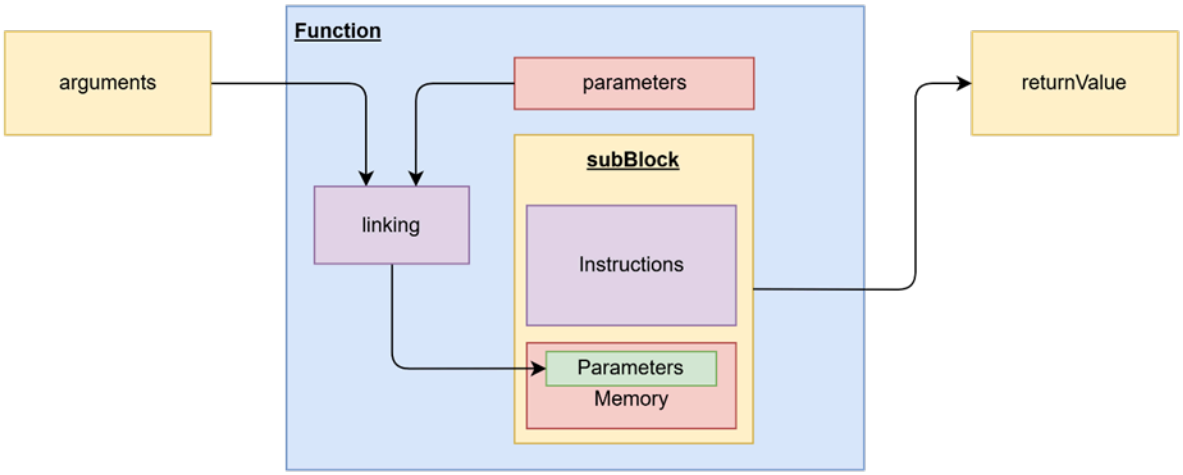


schéma du fonctionnement d'une fonction

En sortie, la fonction renvoie directement la returnValue renvoyée par son subBlock.

Les boucles

Dans CTranslater, les boucles sont constituées de deux SubBlocks essentiels qui jouent chacun un rôle spécifique : la condition et les instructions de la boucle. Lors de l'exécution de la boucle, le premier SubBlock est exécuté pour évaluer la condition, et en fonction du résultat, le second SubBlock, contenant les instructions de la boucle, est exécuté ou non. Ce processus se répète tant que la condition reste vraie.

Lorsque la condition de la boucle devient fausse, l'exécution de la boucle s'arrête. La boucle renvoie donc les variables mis à jour au bloc parent pour que ce dernier puisse actualiser ses variables. Cette étape garantit que toutes les modifications apportées pendant l'exécution de la boucle sont prises en compte par l'élément parent.

Il est important de noter que l'exécution d'une boucle peut être interrompue prématurément si une instruction returnValue est rencontrée dans le SubBlock de la boucle. Dans ce cas, la boucle est immédiatement arrêtée et une returnValue, ainsi que le scope, est envoyée au parent.

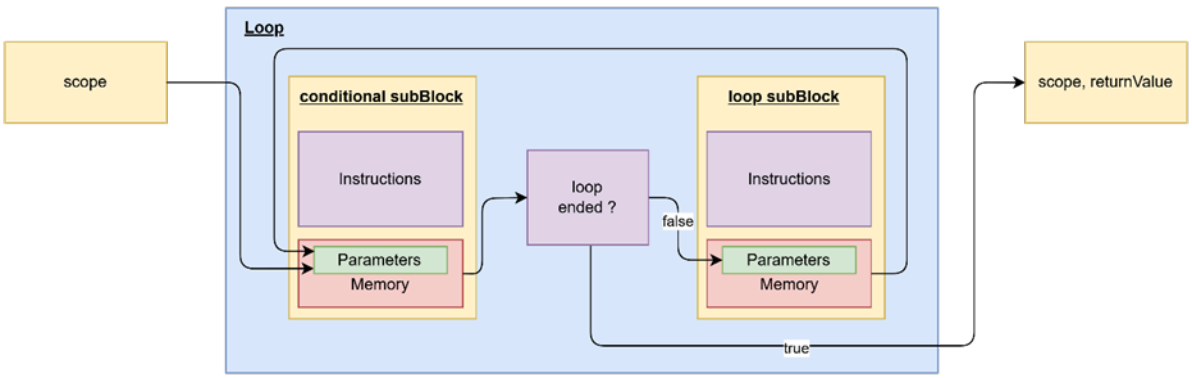


schéma du fonctionnement d'une boucle

L'un des aspects clés de ce fonctionnement est le partage du scope entre la boucle et son parent. En effet, la boucle hérite du scope du parent et peut y accéder et y apporter des modifications. Cela signifie que les variables du parent sont lisibles et modifiables à la fois dans la condition et dans les instructions de la boucle.

En vérité les variables du parents ne sont pas directement modifiables. Il s'agit d'une copie des variables. A la fin de la boucle, la returnValue ainsi que la liste des variables sont retournés au bloc parents. Ainsi, le bloc parents reconnaissant que le subBlock est une boucle va mettre a jour ses propres variables en fonction de celles retournés.

Les instructions conditionnelles

Dans CTranslater, les instructions conditionnelles sont composées de trois subBlocks. Le premier subBlock, appelé conditionalSubBlock, est exécuté en premier. Lorsque ce subBlock est exécuté, il commence par évaluer l'expression conditionnelle. Si l'expression retourne vrai, le programme passe à l'exécution du true_subBlock , qui contient les instructions à exécuter lorsque la condition est remplie. Si l'expression retourne faux, l'exécution se poursuit alors avec le false_subBlock, qui contient les instructions à exécuter dans le cas contraire.

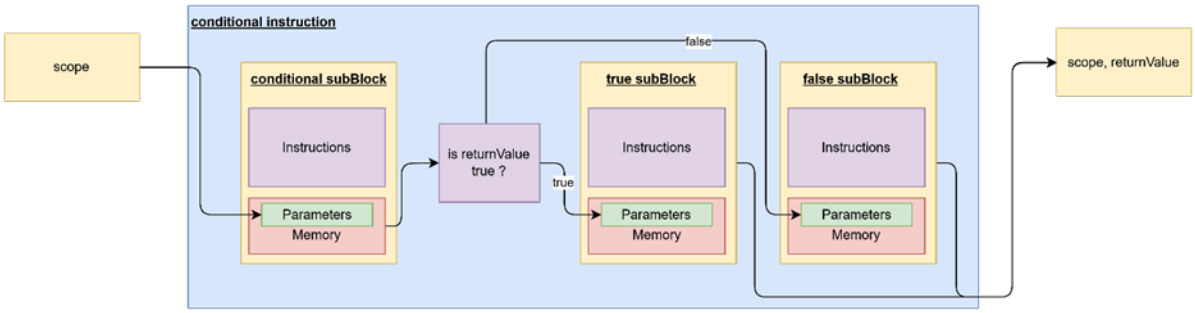


Schéma de la structure d'une instruction conditionnelle

Chaque subBlock, qu'il soit vrai ou faux, partage le scope du parent. Cela permet d'utiliser les variables définies avant la condition ou transmises par le parent dans les deux branches conditionnelles. Si des modifications sont effectuées dans l'une des branches, elles affectent le scope parent, qui est mis à jour après l'exécution grâce aux variables retournés.

Sur le meme principe que pour les boucles, si une instruction returnValue est rencontrée dans un des deux subBlocks, vrai ou faux, l'éxécution est stoppé immédiatement et la pair (liste de variable, returnValue) est renvoyé à l'élément parent.

Génération du code C

Le processus de génération de code dans CTranslater repose sur une structure encadrée. Le code généré par l'utilisateur est automatiquement encapsulé entre un en-tête et un pied de page. Ces deux sections assurent l'initialisation, la gestion correcte de la SDL, la possibilité de se déplacer sur le dessin généré et autres.

L'en-tête ressemble à:

```
#include <SDL2/SDL.h>
#include <stdio.h>

#include "sdlEncapsulation.h"



int main(int argc, char* args[]) {

    SDL_Window *window = NULL;
    SDL_Renderer *renderer = NULL;

    if (initSDL(&window, &renderer) != 0)
    {
        return -1;
    }

    int returnStatement = 0;
    Dpp_Canvas drawCanvas; // Will be filled by beginCanvas

    // Start of the generated code
```

Et le pied de page à :

```
    //End of the generated code

    runCanvasViewer(renderer, window, drawCanvas);

    return returnStatement;
}
```

Le code généré vient donc se glisser au milieu. La génération du code se fait ligne par ligne par une suite d'instructions élémentaires. L'ensemble du code est décomposé en instruction élémentaire, il en est de même pour les instructions complexes. Ainsi, par exemple lors de l'utilisation d'une fonction, le nom de celle-ci n'apparaîtra pas le fichier C généré, mais seulement sa décomposition en instructions élémentaires.

Axe d'amélioration

Bien que fonctionnel, CTranslater peut être amélioré sur plusieurs points. D'abord, la gestion des erreurs pourrait être renforcée en intégrant une détection plus fine des problèmes, avec l'objectif de garantir que « si le code compile, alors il fonctionnera ». Il serait aussi intéressant de fournir des messages d'erreur structurer afin de les remonter à la structure appelante de CTranslater, lui permettant ainsi de gérer l'erreur et notifier l'utilisateur. Ensuite, le code généré pourrait être optimisé: au lieu de traduire ligne par ligne, même pour les boucles, celles-ci pourraient être directement traduites en structures natives du C (for, while), permettant de bénéficier des optimisations offertes par les compilateurs.

Compilation et *linking*

La partie finale, le fichier `pydpp/compiler/toolchain.py`, fait appel au compilateur du système d'exploitation pour compiler le fichier C produit par le CTranslater avec notre librairie `sdlEncapsulation`.

Sur Linux, `cc` est utilisé (ce qui revient à dire `gcc` le plus souvent). La commande suivante est utilisée pour compiler le programme (on assume que SDL2 est installé) :

```
cc fichier_c_généré.c
   libs/sdlEncapsulation/bin/sdlEncapsulation.a
   -o executable
   -I libs/sdlEncapsulation/include
   -lm
   -lSDL2
```

Sur Windows, c'est beaucoup plus compliqué. Un **fichier batch** permet de récupérer le chemin vers le compilateur Visual Studio, grâce à `vswhere.exe`, un outil fourni par Microsoft. Ce fichier donne des variables d'environnement qui sont sauvegardées dans un fichier temporaire pour **accélérer la compilation**. Ensuite, le compilateur `cl` est appelé :

```
cl  -I libs/sdlEncapsulation/include
    -I libs/SDL/include
    fichier_c_généré.c
    libs/sdlEncapsulation/bin/sdlEncapsulation.lib
    libs/SDL/lib/SDL2.lib
    libs/SDL/lib/SDL2main.lib
    shell32.lib
    /MDd
    /Fodossier_du_fichier_c
    /link
    /out:dossier_de_l'executable
    /utf-8
    /MACHINE:X64
    /SUBSYSTEM:CONSOLE
```

Le fichier DLL `SDL2.dll` est aussi copié dans le dossier de l'exécutable.

Module de suggestions

Le module de suggestions (dans `pydpp/compiler/suggestion.py`), permet de proposer automatiquement des **changements sur le code** pour **chaque erreur compatible dans le code**.

Ce module analyse **tous les nœuds erronés** dans l'arbre de syntaxe, et nécessite le résultat de l'analyse sémantique. Si l'erreur possède un code d'erreur connu, l'algorithme va essayer de proposer une **suggestion** pour ce nœud. Une suggestion contient :

- un titre
- une erreur associée
- une fonction permettant d'appliquer le correctif

Le cœur d'une suggestion réside dans sa **fonction d'application**. Cette fonction va **modifier l'arbre de syntaxe** pour ajouter, supprimer ou remplacer des nœuds. Ces changements seront **retraduits en code sous format texte** en parcourant **tous les *tokens* de l'arbre**.

Prenons, par exemple, la suggestion proposée lorsqu'un point-virgule est manquant :

```
def find_suggestion(node, semantic_info, problem) → Suggestion | None:
    match problem.code:
        case ProblemCode.MISSING_SEMICOLON:
        def apply(n: Node):
            assert isinstance(n, Statement)
            n.semi_colon = leaf(Token(TokenKind.SYM_SEMICOLON, ';'))
            return True

        return Suggestion("Ajouter un point-virgule", problem, apply)
```

Ici, on remplace l'emplacement `semi_colon` avec un nouveau **nœud feuille** composé d'un *token* de point-virgule, `SYM_SEMICOLON`.

Moteur de rendu de dessin

Nous avons utilisé SDL2, un outil d'imagerie 2D multi-platforme. Nous avons créé une bibliothèque qui utilise SDL2, nommée `sdlEncapsulation`, pour pouvoir fournir des **fonctions de dessin** de différentes formes, ainsi qu'un visionneur d'image générée. En passant par une librairie, on évite ainsi de recompiler l'ensemble de ces fonctions à chaque compilation du code généré.

Dessin des formes

Les formes peuvent prendre en compte **la couleur, l'angle, et l'épaisseur** du curseur, et chaque fonction en C correspond à une forme différente (creuse ou pleine).

Ainsi, il est possible de dessiner :

- un rectangle creux et plein avec rotation (`drawRectangleOutline`/`drawRectangleFill`)
- un cercle creux et plein (`drawCircleOutline`/`drawCircleFill`)
- une ligne (`drawThickLine`)
- un pixel (`drawPixel`)

SDL nous permettant uniquement de remplir les pixels de l'image manuellement, ou de remplir des triangles, nous avons dû créer des **algorithmes spéciaux** pour dessiner certaines formes. Notamment :

- **cercle creux** : avec un rayon r et d'épaisseur t, on vérifie l'**équation de la distance euclidienne**

  $$r - t \leq x^2 + y^2 \leq r$$

  sur tous les pixels d'un carré de taille r
- **cercle plein** : même algorithme que le cercle creux avec $t = r$, donc $x^2 + y^2 \leq r$
- **rectangle plein** : ayant son coin inférieur gauche situé à $S = (x, y)$ de largeur L et une longueur l, on remplit **deux triangles issus des quatre points du rectangle** ; la rotation d'angle α est réalisée en appliquant la matrice de rotation associée à α sur chaque point P du rectangle, translaté de sorte que S soit à l'origine :

  $$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} (P - S) + S$$
- **rectangle creux** : similaire au rectangle plein, mais on dessine des **lignes** au lieu de dessiner des triangles
- **lignes** : allant de $u = (x_1, y_1)$ à $v = (x_2, y_2)$ et d'épaisseur t, une ligne est dessinée à l'aide de **deux triangles créés de quatre points définis** par cette formule :

$$D = \frac{v-u}{\|v-u\|} \quad M = \frac{t}{2}\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} D$$

$$P_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + M \quad P_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - M \quad P_3 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + M \quad P_4 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} - M$$

Visionneur d'image

Une fois le programme Draw++ terminé, il serait très pratique de pouvoir se déplacer et zoomer sur l'image facilement. C'est ce que permet le **visionneur d'image**, qui ne nécessite que deux fonctions C : `beginCanvas` et `runCanvasViewer`.

Lors de l'exécution du programme, le code généré par le compilateur appelle la fonction `beginCanvas` qui va **créer une texture** de la taille voulue et **rediriger tous les dessins sur cette texture**, plutôt que de dessiner directement à l'écran.

Ensuite, `runCanvasViewer` va lire cette texture pour fournir une interface simple permettant de **voir l'image** comme on le souhaite. Il est possible de **déplacer l'image** en glissant avec la souris ou en utilisant les flèches directionnelles, et de **zoomer** avec la mollette.

Lors de l'initialisation, le programme calcule automatiquement les coordonnées nécessaires pour **centrer et recadrer l'image** de sorte qu'elle soit **entièrement visible à l'écran**.