Compilation et interprétation
IFT-3101

Énoncé TP2

Analyse syntaxique, sémantique et production de code intermédiaire

Directives :

- Date de remise : Dimanche 8 décembre à 23h59. Tous les retards sont acceptés **sans pénalité** jusqu'au lundi 9 décembre à 7h59. La note de 0% vous sera automatiquement attribuée après 8h si vous n'avez pas remis votre travail.
- Il s'agit d'un travail individuel. On vous encourage à collaborer entre-vous, tant que vous remettez votre propre travail. Les sanctions habituelles concernant le plagiat s'appliquent.
- Vous devez compléter le code *C#* qui vous est fourni. Vous ne pouvez pas modifier la signature des méthodes fournies, mais vous êtres libres d'ajouter autant de méthodes privées que vous le souhaiter.
- Aucune librairie externe n'est autorisée. Vous êtes limités à la librairie standard *C#*.
- La pondération pour ce travail est de 20%. Les critères de correction sont présentés en annexe A.

Contexte :

Pour ce travail pratique, vous devez compléter l'implémentation de méthodes utilitaires pour le front-end d'un compilateur. Le front-end permet de lire une séquence de tokens et peut produire un arbre d'analyse, calculer les valeurs des attributs ou encore produire du code intermédiaire. Des tests fonctionnels sont fournis pour valider l'implémentation de vos méthodes. Quatre scénarios sont présentés dans le fichier `FunctionalTests.cs` :

- `Exercises1D` :
  Scénario qui reprend l'exercice (1.d) du module sur le Top-Down parsing. Ce scénario vérifie que les symboles de la grammaire sont bien identifiés, que les ensembles First et Follow sont bien calculés et que la table d'analyse produit le résultat attendu.
- `SyntaxArithmeticGrammar` :
  Scénario basé sur l'exemple utilisé en classe tout au long du module sur le Top-Down parsing. Le scénario valide encore une fois les ensembles First et Follow et le contenu de la table d'analyse. De plus, ce scénario vérifie que l'arbre d'analyse produit pour une certaine séquence correspond à l'arbre désiré.
- `SemanticArithmeticGrammar` :
  Ce scénario introduit le calcul des attributs sur les noeuds de l'arbre d'analyse. La grammaire attribuée est la même que celle utilisée en classe pour incorporer des attributs synthétisés et hérités. Le scénario vise principalement à comparer les valeurs des attributs sur chacun des noeuds.

- `SyntaxArithmeticGrammar` :
  Le dernier scénario introduit des attributs spécifiques à la production de code intermédiaire. La grammaire utilisée est une adaptation simplifiée de la grammaire étudiée en classe à travers les modules sur la production de code intermédiaire. Les vérifications de ce scénario concernent principalement le contenu de l'attribut ultime de la racine. Voici la SDD adaptée :

	Production	Règles sémantiques
(1)	$S \rightarrow$ `if` $B\ M_0\ S'_0$ `else` $M_1\ S'_1$	$Backpatch(B.tl, M_0.id)$ $Backpatch(B.fl, M_1.id)$ $S.code = B.code + S'_0.code + S'_1.code$
(2)	$B \rightarrow B'\ R$	$B.tl = R.tl$ $B.fl = R.fl$ $R.itl = B'.tl$ $R.ifl = B'.fl$ $B.code = B'.code + R.code$
(3)	$B' \rightarrow$ `id`$_0$ `op` `id`$_1$	$B'.tl = MakeList(nextId)$ $B'.fl = MakeList(nextId + 1)$ $B'.code =$ "`if` `id`$_0.val$ `op`$.val$ `id`$_1.val$ `goto` _" + "`goto` _"
(4)	$R \rightarrow \|\|\ M\ B$	$R.tl = Merge(R.itl, B.tl)$ $R.fl = B.fl$ $Backpatch(R.ifl, M.id)$ $R.code = B.code$
(5)	$R \rightarrow \&\&\ M\ B$	$R.tl = B.tl$ $R.fl = Merge(R.ifl, B.fl)$ $Backpatch(R.itl, M.id)$ $R.code = B.code$
(6)	$R \rightarrow \epsilon$	$R.tl = R.itl$ $R.fl = R.ifl$ $R.code =$ " "
(7)	$M \rightarrow \epsilon$	$M.id = nextId$
(8)	$S' \rightarrow$ `id = nb`	$S'.code =$ "`id`$.val$ `= nb`$.val$"

Pour la séquence

```
if ( x < 100 || x > 200 && x != y ) a = 4 else a = 8
```

le code intermédiaire produit devrait être constitué des 8 instructions suivantes :

```
100.  if x < 100 goto 106
101.  goto 102
102.  if x > 200 goto 104
103.  goto 107
104.  if x != y goto 106
105.  goto 107
106.  a = 4
107.  a = 8
```

Vous n'avez pas à modifier les tests fonctionnels. Votre tâche pour ce travail est de compléter le code des méthodes spécifiées plus bas dans le but de satisfaire l'entièreté des tests.

Tâches :

0. **Mise en place de l'environnement de développement**

   Vous avez à votre disposition une trentaine de fichiers de code *C#* ainsi que les éléments nécessaires pour créer une solution Visual Studio. Le projet est également accompagné de tests unitaires reposant sur le framework NUnit. Vous pouvez sans problème ignorer les fichiers `*.sln` et `*.csproj` et vous tourner vers l'IDE de votre choix.

   Le code qui vous est fourni a été conçu avec .NET 8.0 et la version *C#* 12.0.

1. **Gestion des erreurs**

   Chaque classe est jumelée avec des tests unitaires lorsque certains cas spéciaux demandent une attention particulière. Pour chacun des cas énumérés ci-bas, vous devez transcrire le scénario dans le test mentionné et ensuite compléter le code pour valider votre implémentation.

   Les tests ne doivent pas nécessairement être implémentés dans l'ordre. Vous pouvez compléter la prochaine section avant de revenir aux tests. Certains tests ne peuvent pas être satisfaits avant d'ajouter le code nécessaire au fonctionnement générique de la méthode visée.

   (a) Classe `Token`

   - `WhenSymbolIsNonterminal` :
     Par définition, un token représente un symbole terminal. Le constructeur doit lancer une exception si le symbole associé est un non-terminal.

   - `WhenSymbolIsSpecialOtherThanEnd` :
     On permet également le symbole spécial *end* d'être représenté par un token. Si un symbole spécial autre que *end* est référencé, le constructeur doit lancer une exception.

   (b) Classe `Production`

   - `WhenHeadIsTerminal` :
     La tête d'une production devrait toujours être un symbole non-terminal. Si la tête de la production est un symbole terminal, le constructeur doit lancer une exception.

   - `WhenHeadIsSpecial` :
     Pour la même raison, le constructeur doit lancer une exception si la tête de la production est un symbole spécial.

- `WhenBodyContainsSpecialOtherThanEpsilon` :
  Le corps d'une production peut contenir n'importe quelle combinaison de symboles terminaux et non-terminaux ou bien le symbole spécial *epsilon*. Si le corps de la production contient d'autres symboles spéciaux, le constructeur doit lancer une exception.

- `WhenBodyWithMoreThanOneSymbolContainsEpsilon` :
  Le symbole spécial *epsilon* n'est autorisé dans le corps de la production que lorsqu'il s'agit du seul symbole. Le constructeur doit lancer une exception si le symbole *epsilon* est accompagné d'autres symboles, peut importe sa position.

(c) Classe `ParseNode`

- `WhenSymbolIsNonterminalAndProductionIsNull` :
  La classe `ParseNode` représente un noeud de l'arbre d'analyse obtenu lors de la l'analyse syntaxique. Un noeud est associé à un symbole et une production. Un noeud jumelé à un symbole non-terminal doit posséder des enfants correspondant au corps de la production. Le constructeur doit donc lancer une exception si le symbole est non-terminal et que la production est *null*.

- `WhenSymbolIsNonterminalAndIsNotHeadOfProduction` :
  Pour les mêmes raisons, le constructeur doit également lancer une exception si le symbole du noeud n'est pas le même que celui de la tête de la production.

- `WhenSymbolIsTerminalAndProductionIsNotNull` :
  Inversement, la production doit être *null* lorsque le noeud ne représente pas un symbol non-terminal. Le constructeur doit lancer une exception si le symbole est terminal et que la production n'est pas *null*.

- `WhenSymbolIsEpsilonAndProductionIsNotNull` :
  De manière similaire, le constructeur doit également lancer une exception si le symbole référencé est *epsilon*, mais que la production n'est pas *null*.

- `WhenSymbolIsSpecialOtherThanEpsilon` :
  Les autres symboles spéciaux ne sont pas autorisés dans un `ParseNode`. Le constructeur doit finalement lancer une exception si un symbole spécial autre que *epsilon* est référencé.

- `WhenNoValueForAttribute` :
  La méthode `GetAttributeValue` doit lancer une exception si aucune valeur ne peut être trouvée pour l'attribut spécifié.

- `WhenNodeCannotBeFound` :
  La méthode `GetBindedNode` doit lancer une exception s'il est impossible de récupérer le noeud recherché parmi le noeud lui-même et ses enfants directs.

(d) Classe `SemanticAction`

- `WhenTargetIsInSources` :
  La classe `SemanticAction` englobe une action sémantique qui sera exécutée lors

du calcul des attributs. Une instance de cette classe conserve des références sur l'attribut cible et les attributs sources.

Pour une règle sémantique $A.x = B.y + C.z$, l'attribut $A.x$ est la cible tandis que $B.y$ et $C.z$ sont les sources. Pour des raisons évidentes, le constructeur doit lancer une exception lorsque la cible se retrouve également dans les sources.

- `WhenNodeSymbolIsNonterminalAndTargetIsNotInNode` :
  La méthode `Execute` doit lancer une exception si la cible ne peut pas être trouvée dans le noeud référencé. Pour un noeud qui représente le symbole non-terminal A et la production $A_0 \rightarrow a_0 A_1$, il est possible de rencontrer une cible $A_0.x$ ou bien $a_0.y$, mais pas $A_2.x$ ni $B_0.x$.

- `WhenNodeSymbolIsNonterminalAndSourceIsNotInNode` :
  Pareillement, la méthode `Execute` doit également lancer une exception dès qu'un des attributs sources ne peut pas être atteint dans le noeud ou ses enfants, toujours dans le cas où le symbole du noeud est un symbole non-terminal.

- `WhenNodeSymbolIsTerminal` :
  Il n'est pas possible d'exécuter une action sur un noeud associé à un symbole terminal, car seuls les attributs synthétisés provenant de l'analyseur lexical sont autorisés sur les symboles terminaux. La méthode `Execute` doit lancer une exception si un tel cas se produit.

- `WhenNodeSymbolIsSpecial` :
  Finalement, la méthode `Execute` doit aussi lancer une exception si le symbole du noeud est un symbole spécial, puisque ces symboles ne contiennent aucun attribut.

(e) Classe `SyntaxDirectedTranslationScheme`

- `WhenStartSymbolIsTerminal` :
  Le symbole de départ de la grammaire ne peut pas être un symbole terminal. Le constructeur doit lancer une exception lorsque cette situation se produit.

- `WhenStartSymbolIsSpecial` :
  Le constructeur doit aussi lancer une exception si le symbole de départ est un symbole spécial.

- `WhenStartSymbolIsNotDefined` :
  Le symbole de départ doit évidemment se retrouver dans la tête d'au moins une production. Si ce n'est pas le cas, le constructeur doit lancer une exception.

- `WhenNonterminalIsNotDefined` :
  Le corps d'une production peut contenir n'importe quelle combinaison de symboles terminaux et non-terminaux. Chaque symbole non-terminal doit se retrouver dans la tête d'au moins une production. Le constructeur doit lancer une exception si un symbole non-terminal n'est pas défini dans les productions.

- `WhenNoTerminaInProductions` :
  Le constructeur doit lancer une exception s'il n'y a aucun symbole terminal dans l'ensemble des productions.

- `WhenProductionsAreEquivalent` :
  Deux productions sont considérées équivalentes si le même symbole non-terminal se trouve à la tête des deux productions et que leur corps contiennent la même séquence de symboles. Le constructeur doit lancer une exception si des productions équivalentes sont identifiées.

- `WhenDefinitionIsNotLAttributed` :
  Chaque production est accompagnée d'un ensemble de règles sémantiques, possiblement vide. Le constructeur doit lancer une exception si la définition ne respecte pas les contraintes d'une grammaire L-attribuée.

- `WhenInputIsEmpty` :
  La méthode `FirstOfBody` doit lancer une exception si la liste de symboles reçue est vide.

- `WhenInputSymbolIsNotInGrammar` :
  La méthode `FirstOfBody` doit également lancer une exception si un des symbole de la liste reçue ne fait pas partie de la grammaire.

(f) Classe `LLParsingTable`

- `WhenFirstFirstConflict` :
  Le constructeur doit lancer une exception lorsqu'il y a un conflit First/First lors de la construction de la table d'analyse. Par exemple, la grammaire suivante présente un conflit First/First.

$$\begin{array}{lcl} S & \rightarrow & A \mid B \\ A & \rightarrow & a \\ B & \rightarrow & a \end{array}$$

  Pour le symbole non-terminal S et les deux productions $S \rightarrow A$ et $S \rightarrow B$, on obtient $\text{First}(A) \cap \text{First}(B) \neq \emptyset$. Deux entrées seront donc placées dans la cellule $[S, a]$ de la table d'analyse.

- `WhenFirstFollowConflict` :
  Le constructeur doit lancer une exception si un conflit First/Follow est détecté pendant la construction de la table d'analyse. Par exemple, la grammaire suivante apporte un conflit First/Follow.

$$\begin{array}{lcl} S & \rightarrow & A\ a \\ A & \rightarrow & a \mid \epsilon \end{array}$$

  Pour le symbole non-terminal A, puisque $\epsilon \in \text{First}(A)$ et $\text{First}(A) \cap \text{Follow}(A) \neq \emptyset$, il y aura deux entrées dans la table d'analyse pour la cellule $[A, a]$.

- `WhenNonterminalIsTerminal` :
  La méthode `GetProduction` doit lancer une exception si le symbole non-terminal

est un symbole terminal.

- `WhenNonterminalIsSpecial` :
  De la même manière, la méthode `GetProduction` doit lancer une exception si le symbole non-terminal est un symbole spécial.

- `WhenTerminalIsNonterminal` :
  Inversement, la méthode `GetProduction` doit aussi lancer une exception si le symbole terminal est un symbole non-terminal.

- `WhenTerminalIsSpecialOtherThanEnd` :
  Le symbole spécial *end* est autorisé dans la table d'analyse. La méthode `GetProduction` doit toutefois lancer une exception si le symbole terminal est un symbole spécial autre que le symbole *end*.

- `WhenNonterminalIsNotInTable` :
  La méthode `GetProduction` doit lancer une exception si le symbole non-terminal ne fait pas partie de la table d'analyse.

- `WhenTerminalIsNotInTable` :
  Finalement, la méthode `GetProduction` doit aussi lancer une exception si le symbole terminal ne peut pas être trouvé dans la table d'analyse.

(g) Classe `TopDownParser`

- `WhenEndTokenIsNotAtEndOfInput` :
  La méthode `Parse` produit une arbre d'analyse à partir d'une séquence de tokens reçue en entrée. Le symbole *end* permet de marquer la fin de la séquence. Cette méthode doit lancer une exception si le dernier token de la séquence d'entrée ne correspond pas au symbole *end*.

- `WhenCannotMatchInput` :
  La méthode `Parse` doit lancer une exception si pendant l'exécution, le symbole terminal sur le dessus de la pile ne correspond pas au prochain symbole de l'entrée.

- `WhenNoProductionDefined` :
  La méthode `Parse` doit également lancer une exception si la table d'analyse ne contient aucune production pour le symbole non-terminal sur le dessus de la pile et le prochain symbole de l'entrée au courant de l'exécution.

(h) Classe `IntermediateCodeManager`

- `WhenInstructionIsAlreadyRegistered` :
  La méthode `RegisterForBackpatching` permet à une instruction de s'enregistrer pour éventuellement bénéficier du *backpatching*. Une même instruction ne peut pas s'enregistrer plus d'une fois. La méthode doit lancer une exception si une instruction s'est déjà enregistrée.

- `WhenLabelIdIsSmallerThanFirstId` :
  La méthode `Backpatch` applique le `labelId` spécifié à toutes les instructions de la liste. Le `labelId` reçu doit évidemment être valide. La méthode doit lancer une exception si le `labelId` est inférieur à la valeur initiale.

- `WhenLabelIdIsGreaterOrEqualThanNextId` :
  Dans le même ordre d'idée, il est impossible d'associer un `labelId` qui n'a pas encore été produit à une instruction. La méthode `Backpatch` doit lancer une exception si le `labelId` est supérieur ou égal à la prochaine valeur.

- `WhenJumpLabelIsNotNull` :
  Il ne devrait pas être possible de modifier un `JumpLabel` déjà initialisé. La méthode `Backpatch` doit finalement lancer une exception si le `JumpLabel` d'une instruction de la liste n'est pas *null*.

Note : Il est normal que le code des tests unitaires soit très répétitif. Pensez à utiliser le ***`Setup`*** *des classes de tests lorsque possible.*

2. **Implémentation de la logique**

La prochaine étape est d'implémenter les méthodes spécifiées de chacune des classes listées plus bas. Ces classes on pour objectif de permettre à une séquence de tokens d'être convertie en arbre d'analyse, calculer les attributs sur les noeuds de l'arbre et finalement produire du code intermédiaire valide.

Pour alléger la présentation, la gestion des erreurs est ignorée dans cette section. Référez-vous à la section précédente pour en apprendre davantage sur les cas d'utilisation non supportés. Vous devez incorporer la gestion des erreurs dans le code à compléter pour la logique des méthodes de cette section.

(a) Classe `ParseNode`

Cette classe représente un noeud de l'arbre d'analyse. Chaque noeud est associé à un symbole et une production. Les enfants d'un noeud représentent les symboles du corps de la production. Les valeurs des attributs sont sauvegardés au niveau du noeud.

- `GetBindedNode` :
  Cette méthode reçoit en paramètre un `AttributeBinding`, qui contient de l'information sur l'attribut du noeud recherché. Cette méthode ne manipule pas les attributs, mais permet plutôt de parcourir le noeud et ses enfants directs pour retrouver le noeud qui correspond au `AttributeBinding`.

  Pour un noeud associé au symbole A ayant trois enfants a, A et B, le noeud A_0 correspond au noeud lui-même, tandis que a_0 et A_1 représentent respectivement le premier et le deuxième enfant.

Note : Il suffit de compter le nombre de fois que chaque symbole est rencontré, sans oublier le symbole du noeud en question.

(b) Classe `SyntaxDirectedTranslationScheme`

La classe `SyntaxDirectedTranslationScheme` représente un SDT. La classe contient également les informations concernant la grammaire et les actions sémantiques associées à chaque production.

- `SetSymbols` :
  Cette méthode privée doit simplement extraire les symboles terminaux et non-terminaux des règles de production pour construire les listes `Terminals` et `Nonterminals` du SDT.

- `OrderRules` :
  La méthode privée `OrderRules` a pour mission d'ordonner les actions sémantiques selon l'ordre des symboles de la production et des dépendances entre les attributs. Il s'agit en quelques sortes de la conversion entre une SDD et un SDT.

  Pour une production $A \to aAB$ et les actions sémantiques $B_0.y = 1$, $A_1.z = A_1.x$, $A_0.x = a_0.x + B_0.y$ et $A_1.x = a_0.x$, l'ordre final des actions pour cette production devrait être d'abord $A_1.x = a_0.x$, ensuite $A_1.z = A_1.x$ suivi de $B_0.y = 1$ et finalement $A_0.x = a_0.x + B_0.y$.

  En d'autres termes, pour une production $A \to BC$, on devrait trouver d'abord les actions qui ciblent un attribut de B, ensuite les actions qui ciblent un attribut de C, et finalement les actions qui ciblent un attribut de A. Si plusieurs actions ciblent le même symbole, ces dernières sont ordonnées en fonction de leurs sources.

  L'ordonnancement des actions sémantiques est indépendant pour chaque production. Le résultat devrait se retrouver dans le dictionnaire `Rules` de l'objet.

- `ComputeFirstSets` :
  Cette méthode privée doit calculer les ensembles First de chaque symbole non-terminal de la grammaire et conserver le résultat dans le dictionnaire `First` de l'objet. Les ensmbles First ne contiennent que des symboles terminaux ou bien le symbole *epsilon*.

  L'ensemble First de chaque symbole non-terminal est initialement vide. Une boucle englobe la découverte des symboles selon les règles et propriétés vues en classe. Cette boucle se répète tant qu'au moins un symbole est ajouté à un ensemble.

- `ComputeFollowSets` :
  De manière similaire, cette méthode privée détermine l'ensemble Follow de chaque symbole non-terminal de la grammaire. Les ensembles finaux sont sauvegardés dans le dictionnaire `Follow` de l'objet. Seuls les symboles terminaux et le symbole *end* sont autorisés dans les ensembles Follow.

Tout comme pour le calcul des ensembles First, l'ensemble Follow de chaque symbole non-terminal est initialement vide. La découverte des symboles des ensembles Follow se fait également de manière itérative. La boucle s'arrête lorsque aucun changement n'est détecté entre deux itérations.

- `FirstOfBody` :
  La méthode publique `FirstOfBody` doit simplement retourner l'ensemble First qui correspond à la liste de symboles reçu en paramètre. Le symbole *epsilon* ne peut faire partie du résultat que si l'entièreté de l'entrée peut être dérivée pour obtenir le mot vide.

Note : Les méthodes peuvent paraître initialement complexes, mais beaucoup de code se répète et peut être réutilisé. Pensez également à recycler les méthodes déjà existantes.

(c) Classe `LLParsingTable`

Sans surprise, cette classe contient la logique derrière la construction et la consultation d'une table d'analyse.

- `Constructor` :
  Le constructeur de la classe `LLParsingTable` doit directement remplir la table d'analyse du SDT obtenu en paramètre selon les propriétés et techniques vues en classe.

  La table d'analyse est un dictionnaire qui contient comme clés les symboles non-terminaux. À chaque symbole non-terminal est associé un dictionnaire de symboles terminaux (ou bien le symbole *end*) et les valeurs finales sont des productions. Une cellule vide de la table d'analyse est représentée par *null*.

Note : Le SDT partagé à la construction de l'objet est très riche en information, utilisez-le judicieusement.

(d) Classe `TopDownParser`

Cette classe conserve une référence sur une table d'analyse et permet de construire un arbre d'analyse à partir d'une séquence de tokens selon les techniques de Top-Down parsing. Cet arbre d'analyse, constitué de `ParseNodes`, représente la structure de l'entrée en fonction de la grammaire étudiée par la table d'analyse.

- `Parse` :
  La méthode `Parse` est une implémentation de l'algorithme *Nonrecursive Predictive Parsing*. Cette méthode doit retourner un noeud racine qui représente le symbole de départ. Les noeuds enfants du noeud racine sont les noeuds qui représente les symboles du corps de la première production appliquée, qui contiendront eux-mêmes des enfants selon leur production respective, et ainsi de suite.

L'algorithme manipule une pile de symboles qui est initialisée avec le symbole spécial *end* et le symbole de départ. Lorsqu'un symbole non-terminal est empilé, sa production est inconnue, car l'analyse se fait du haut vers le bas de l'arbre. Le noeud de ce symbole pourra être instancié au moment où sa production sera déterminée en consultant la table d'analyse.

Note : L'algorithme est relativement simple si vous conservez des structures propres et bien ordonnées. Soyez patients, n'essayez pas de développer l'arbre trop rapidement.

(e) Classe `SemanticAnalyzer`

La responsabilité principale de cette classe est de calculer les valeurs des attributs sur un arbre d'analyse.

- `Annotate` :
  Cette méthode représente la traduction indirecte telle que vue en classe. Il s'agit d'une méthode récursive qui annote d'abord les noeuds enfants avant d'annoter noeud en question. Les actions sémantiques doivent être exécutées dans l'ordre spécifié par le SDT.

  Si le noeud représente un symbole terminal, le seul attribut synthétisé possible pour ce noeud est la valeur lexicale du symbole. L'analyseur lexical fourni cette valeur sur demande.

  Pour un noeud qui représente le symbole non-terminal A et la production $A_0 \rightarrow B_0\ a_0\ A_1$, la méthode `Annotate` doit d'abord calculer les attributs hérités de B_0 avant d'annoter le premier noeud enfant. Il faut ensuite annoter le deuxième noeud enfant, qui n'a pas d'autres attributs à calculer puisqu'il représente un symbole terminal. On peut alors exécuter les actions sémantiques qui ciblent les attributs hérités de A_1, suivi de l'annotation du dernier noeud enfant. Finalement, la dernière étape est de calculer les attributs synthétisés de A_0, c'est-à-dire le noeud en question.

Note : N'oubliez pas que chaque noeud est en mesure d'identifier le noeud ciblé par une action sémantiques, à condition qu'il s'agisse de lui-même ou d'un de ses enfants directs.

(f) Classe `IntermediateCodeManager`

Cette classe orchestre la manipulation des instructions du code intermédiaire généré pendant l'analyse sémantique. Il s'agit d'une classe statique qui conserve en mémoire le prochain `Label` à produire. Cette classe est également responsable de la routine entourant le *backpatching*.

La classe `Label` est l'étiquette qui identifie une instruction et le champ `Id` d'une instance de la classe `Label` représente son numéro (100, 101, 102, ...).

- `EmitLabel` :
  Cette méthode est appelée lors de la création de chaque instruction et retourne le prochain `Label`. La méthode doit également incrémenter `_nextId`, qui sera utilisé lors de la création de la prochaine instruction.

  Il peut aussi être utile de conserver dans une structure interne les `Labels` qui ont été produits. Ces derniers pourraient être référencés plus tard lors du *backpatching*.

- `RegisterForBackpatching` :
  Cette méthode est utilisée par les instructions de type `Jump`. Ces instructions contiennent un champ `JumpLabel` (qui correspond au *goto*) dont la valeur sera connue plus tard dans l'analyse.

  La méthode `RegisterForBackpatching` doit conserver dans une structure interne de l'information concernant l'instruction qui désire s'enregistrer.

- `Backpatch` :
  Certaines actions sémantiques peuvent appliquer le *backpatching* sur une liste pour un `labelId` précis. La méthode doit d'abord retrouver les `Labels` des instructions qui se sont préalablement enregistrées à l'aide de la liste reçue en entrée.

  Il est ensuite possible d'initialiser le `JumpLabel` de ces instructions avec le `Label` qui correspond au `labelId` fourni en paramètre.

Note : Les structures internes n'ont pas besoin d'être très sophistiquées. Assurez-vous de travailler avec des instructions de type `Jump` pour avoir accès au champ `JumpLabel`.

3. ***Bonus facultatif*** **: Traduction directe Top-Down**

Dans le code qui vous est fourni, le calcul des attributs est fait exclusivement de manière indirecte. Pour une certaine séquence de tokens, on construit d'abord explicitement un arbre d'analyse avant d'annoter cet arbre en calculant les valeurs des attributs.

Vous devez ajouter toute la logique nécessaire pour implémenter la traduction directe. Cet algorithme n'utilise pas la classe `ParseNode`. Le calcul des attributs et l'analyse de la structure de l'entrée se font de manière implicite dans la pile pendant l'exécution.

Votre algorithme principal doit recevoir une séquence de tokens et produire en sortie une structure qui contient les valeurs des attributs du symbole de départ. En d'autres termes, le résultat de la traduction directe doit correspondre aux attributs retrouvés sur le noeud racine de l'arbre d'analyse annoté par la traduction indirecte pour la même séquence d'entrée.

Ajoutez également au moins un scénario de test fonctionnel afin de valider votre approche. Ce test devrait illustrer le fonctionnement de l'algorithme et comparer le résultat obtenu

avec celui attendu.

Vous pouvez prendre toutes les libertés cératives que vous jugez nécessaires, à condition de ne pas vous simplifier le travail pour les autres tâches à réaliser.

Note : Prenez le temps de réfléchir aux objets qui seront manipulés dans la pile. Encore une fois, la clé est de bien maîtriser les opérations faites sur la pile.

Remise :

Vous devez remettre dans la boîte de dépôt un fichier `*.zip` qui contient l'entièreté de votre travail, y compris les fichiers que vous n'avez pas modifiés et tout le matériel pertinent pour vos tests.

Assurez-vous de remettre seulement le code sans les répertoires résiduels `bin`, `obj`, `.vs`, `.idea`, etc.

Vous devez également ajouter à votre remise le fichier `remise.txt` où vous indiquerez la version .NET et *C#* utilisée.

Bon travail !

Annexe A : Critères de correction.

Votre note finale pour ce travail est la somme des notes de chaque tâche, pour un minimum de 0/20 et un maximum de 22/20.

Tâche	Min	Max	Critère	Pénalité
Tests	-2	8	Pour chaque test unitaire incomplet	-0.5
Node	0	1	`GetBindedNode` ne retourne pas le bon noeud	-1
			Gère incorrectement les noeuds enfants	-1
SDT	0	3	Symboles de la grammaire incorrects	-1
			Ordre des actions sémantiques incorrect	-1
			Ensembles First incorrects	-1
			Ensembles Follow incorrects	-1
			La méthode `FirstOfBody` ne produit pas le bon résultat	-1
Table	0	2	La table d'analyse n'a pas le bon format	-2
			Construction erronée	-2
Parse	0	2	Manipulations invalides sur la pile	-2
			Arbre d'analyse incorrect	-2
Annotate	0	2	Les actions ne sont pas exécutées dans le bon ordre	-2
			Les noeuds enfants ne sont pas annotés dans le bon ordre	-2
BP	0	2	`EmitLabel` ne produit pas un `Label` valide	-1
			Les structures internes ne permettent pas d'appliquer correctement le *backpatching*	-2
Bonus	0	2	Bonus facultatif non complété	-2
			Construit explicitement un arbre d'analyse	-2
			La solution proposée n'est pas très propre	-1
Général	-22	0	Le code remis ne compile pas	-22
			Modifications non autorisées	-22
			Non respect des normes et conventions *C#*	-5
			Le code remis est de mauvaise qualité	-5
			Répertoires résiduels inclus dans la remise	-2
			Fichier `remise.txt` absent ou non complété	-1