Questions de Cours

1- L'analyse lexicale est la première phase de la chaîne de compilation, elle permet la conversion d'une chaîne de caractères en une liste de symboles (unité lexicale, token) comme un nombre, un identificateur ou un point. La spécification de l'analyse lexicale se fait grâce aux expressions régulières. (Fournit les unités lexicales à l'analyseur syntaxique)
L'analyseur syntaxique consiste à mettre en évidence la structure d'un programme, conformément aux règles d'une grammaire formelle. Un analyseur syntaxique est spécifié sous la forme d'une grammaire dont les règles de production peuvent contenir des actions sémantiques. (Fournit l'arbre abstrait à l'analyseur sémantique)
En compilation, **l'analyse sémantique** est la phase intervenant après l'analyse syntaxique et avant la génération de code. Elle effectue les vérifications nécessaires à la sémantique du langage de programmation considéré, ajoute des informations à l'arbre syntaxique abstrait et construit la table des symboles. Les vérifications réalisées par cette analyse sont : la résolution des noms, la vérification des types, l'affectation définitive, la génération du code.

2- Les objectifs principaux de la spécification formelle de la sémantique d'un langage :
- Modéliser la sémantique avec des outils mathématiques
- Atteindre la qualité de la modélisation de la syntaxe
- Etudier la cohérence et la complétude
- Prouver la correction des outils
- Générer automatiquement les outils

3- Hérité (parcours descendant) : calculé avant l'analyse du non terminal
(PS : la récursivité à gauche empêche le parcours descendant !)
Synthétise (parcours ascendant) : calculé pendant l'analyse du non terminal
Les attributs sont divisés en deux groupes : les attributs synthétisés et les attributs hérités. Les attributs synthétisés sont le résultat des règles d'évaluation des attributs ; ils peuvent aussi utiliser les valeurs d'attributs hérités. Les attributs hérités sont passés vers les feuilles à partir des nœuds parents. Dans certaines approches, on utilise les attributs synthétisés pour passer des informations sémantiques vers la racine de l'arbre. De même, les attributs hérités permettent de passer des informations sémantiques vers les feuilles

4- {i -> 1} |- (function j -> i + j) (2) => 3
- Étapes principales lors de l'écriture d'une sémantique opérationnelle pour un
- langage décrit par sa grammaire :
  - Calcul des paramètres
  - Vérifier l'existence d'effets de bords
  - Définir les règles sémantiques pour chaque variante
- Quelles sont les différentes formes de sémantique formelle d'un langage ?
  - Sémantique opérationnelle : Mécanisme d'exécution des programmes
  - Sémantique axiomatique : Mécanisme de vérification des programmes
  - Sémantique translationnelle : Traduction vers un autre langage équipé d'une sémantique formelle
  - Sémantique dénotationnelle : Traduction vers un formalisme mathématique

Sémantique attribuée

L'objectif est de d´efinir une s´emantique attribu´ee pour calculer la valeur d'une fraction rationnelle définie par la grammaire (A, V, F, P) avec l'alphabet A = {c, d, m} (c désigne un chiffre, désigne la division / et m désigne l'opposé -), les non-terminaux V = {F, N, S}, l'axiome F et les règles de production :

P = F →N d N
F →m N d N
N →c S
S →Λ
S →c S

a- Arbre Syntaxique (de dérivation pour -1/16)
b- Proposer des attributs sémantiques pour les non terminaux de V et décorer l'arbre de dérivation donnée à la question précédente (vous ne recopiez pas l'arbre) pour calculer la valeur de la fraction rationnelle -1/16.
· F : valeur : entier
· V : valeur : entier
· S : valeur : entier
profondeur : entier
c- Actions sémantiques pour les règles de production P qui calculent la valeur de la fraction rationnelle.
F.valeur = N.valeur / N.valeur
F.valeur = -(N.valeur / N.valeur)
N.valeur = c.valeur * 10 ^ (S.profondeur) + S.valeur
S.valeur = 0
S.valeur = c.valeur * 10 ^ (S.profondeur) + S.valeur
S.profondeur = S.profondeur + 1

Traitement par cas en miniML

a- Règles de sémantique cas sans erreurs:
b- Règles de sémantique cas avec erreurs:
c- Règles de typage:

```
type ast =
 | ...
 | MatchNode of ast * ((ast * ast) list) * ast
```

d- TYPE: Function ruleMatch

```
let rec type_of_expr expr env = match expr with
 | ...
 | (MatchNode selection choix defualt) -> ruleMatch selection choix default env
(* unify : typeType -> typeType                    *)
(*     -> typeType * bool                          *)
(* unify permet de comparer 2 types (en valuant eventuellement les     *)
(* variables de types presentes dans chacun des 2). La valeur renvoyee *)
(* est le premier type et true s'il y a correspondance                 *)
(* ErrorType et false sinon                        *)
let ruleMatch selection choix defaut env =
let t_selection = (type _of_expr selection env) and t_def = (type_of_expr defaut env) in
if (types_match t_selection t_def choix env) then t_def else ErrorType

let rec types_match type_selection type_def choix env =
match choix with
        | [] -> true
        | (vi, ei) :: q ->
                let t_vi = (type of_expr vi env) and t_ei = (type of expr ei env) in
                let (_, bool1) = unify t_vi type_selection
                and (_, bool2) - unify t_ei type_ def in
        if (bool1 & boo12) then types_match type_selection type_def q env else false
```

```
let ruleMatch selection choix defaut env =
  let e_i = (List.map snd choix) in
  let f expr = (type_of_expr expr env) in
  let e_i_type = (List.map f e_i) in
  let d_type = (type_of_expr defaut env) in
  let e_i_d_type =  d_type::e_i_type in
  if (aux1 selection choix env) && (aux2 e_i_d_type) then d_type else ErrorType

let rec aux1 selection choix env =
  let e_type = (type_of_expr selection env) in
  match choix with
  | [] -> true
  | [(v,_)]::t -> let v_type = (type_of_expr v env) in
  let _type, _bool = unify e_type v_type in
  _bool && (aux1 selection t env)

let u x = unify x

let rec aux2 l =
    match l with
    | [] -> true
    | e1::q1 -> let u x = unify x e1
                in let bools = (List.map snd (List.map u q1 ))
                in let result = List.fold_left (&&) bools
               in result && (aux2 q1)

let rec ruleMatch selection choix defaut env =
  let s_type = (type_of_expr selection env) and d_type = (type_of_expr defaut env) in
  match choix with
  | [] -> true
  | [(v,e)]::t -> let v_type = (type_of_expr v env) and e_type = (type_of_expr e env) in
  let _, _boolV = unify s_type v_type and _, _boolE = unify d_type e_type in
  _boolV && _boolE && (ruleMatch selection t defaut env)
```

VALUE: Function ruleMatch

```
let rec value_of_expr expr env = match expr with
 | ...
 | (MatchNode selection choix default) -> ruleMatch selection choix default env
let rec ruleMatch selection choix defaut env =
  (let eval =
     (value_of_expr selection env)
  in
     (match choix with
     | [] -> value_of_expr defaut env
     | [(v,e)]::t -> match eval with
                 | (ErrorValue _) as result -> result
                 | v -> (value_of_expr e env)
                 | _ -> ruleMatch selection t defaut env))
```

Instruction de choix en Bloc

L'objectif est de compléter le langage Bloc par une instruction de choix selon la valeur d'une expression. Nous ajoutons pour cela les instructions select et case inspir´ees des langages de la famille C (C, C++, C#, Java, etc) selon la syntaxe CUP suivante :

```
Instruction ::= UL_Select UL_Parenthese_Ouvrante Expression:valeur
                UL_Parenthese_Fermante UL_Accolade_Ouvrante ListeChoix:choix Defaut:defaut
                UL_Accolade_Fermante
{:

:} ;
ListeChoix ::= Choix:choix ListeChoix:reste
{:

:}
           |
{:

:} ;
Choix ::= UL_Case Valeur:valeur UL_Deux_Points Bloc:corps
{:

:} ;
Defaut ::= UL_Default UL_Deux_Points Bloc
{:

:}
|
{:

:} ;
```

Contrairement à la sémantique des langages de la famille C (C, C++, C#, Java, etc), lors de la fin de l'exécution du bloc associé à un choix, le flot de contrôle est transféré à la fin de l'instruction de sélection.
Le type de l'expression sur laquelle la sélection est effectuée doit être compatible avec le type des valeurs associées aux choix.
Les valeurs associées aux choix doivent être de type compatible avec les entiers, les caractères, les booléens ou les chaînes de caractères.
Le programme suivant ne contient pas d'erreurs et s'arrête en affichant la valeur 1.

```
test {
        int i = 0;
        switch (i+1) {
                case 0 : {
                        print 0;
                }
                case 1 : {
                        print 1;
                }
                case 5 : {
                        print 5;
                }
                default : {
                        print -1;
                }
        }
}
```

```
/* int i = 0 */
PUSH 1
LOADL 0
STORE (1) 0[SB]

/* switch (i+1) */
/* i = i + 1 */
PUSH 1
LOADL 1
LOAD (1) 0[SB]
SUBR IAdd
STORE (1) 1[SB]

/* case 0 : { print 0; } */
/* if (i == 0) print 0; */
LOAD (1) 1[SB]
LOADL 0
SUBR IEq
JUMPIF (0) jump_case_1
/* print 0 */
LOADL 0
SUBR IOut
JUMP jump_fin

/* case 1 : { print 1; }  */
/* if (i == 1) print 1; */
jamp_case_1
LOAD (1) 1[SB]
LOADL 1
SUBR IEq
JUMPIF (0) jamp_case_5
/* print 1 */
LOADL 1
SUBR IOut
JUMP jump_fin

/* case 5 : { print 5; }  */
/* if (i == 5) print 5; */
jamp_case_5
LOAD (1) 1[SB]
LOADL 5
SUBR IEq
JUMPIF (0) jamp_case_default
/* print 1 */
LOADL 5
SUBR IOut
JUMP jump_fin

/* case default : {print -1} */
jump_case_default
LOADL -1
SUBR IOut
JUMP jump_fin
jump_fin
HALT
```

Vérification et Construction de l'arbre abstrait

a-

```
public class Switch implements Instruction {
 protected Expression valeur;
 protected ArrayList<Choix> choix;
 protected Defaut defaut;
 public Switch(Expression _val, ArrayList<Choix> _choix, Defaut _defaut) {
 this.valeur = _val;
 this.choix = _choix;
 this.defaut = _defaut;
 }
```

b- CUP

```
Instructions ::= Instructions:instructions Instruction:instruction
                    {:
                        instructions.add( instruction );
                        RESULT = instructions;
                    :}
                |
                    {: RESULT = new LinkedList<Instruction>(); :}
                ;
```

Gestion de la table des symboles

a- Traitements nécessaires:

CollectAndBackwardResolve(TDS);

1. recursive call on Expression
2. Iterate over the list of all cases & recursive call on block
3. recursive call on block by default

FullResolve()

b-

```
 @Override
 public boolean collectAndBackwardResolve(HierarchicalScope<Declaration> _scope) {
 Boolean result = true;
 for (int i=0; i++; i<choix.length) {
 result = result && choix.get(i).collectAndBackwardResolve(_scope);
 }
 result = result && this.valeur.collectAndBackwardResolve(_scope) &&
this.defaut.collectAndBackwardResolve(_scope);
 return result;
 }
 @Override
 public boolean fullResolve(HierarchicalScope<Declaration> _scope) {
 Boolean result = true;
 for (int i=0; i++; i<choix.length) {
 result = result && choix.get(i).fullResolve(_scope);
 }
 result = result && this.valeur.fullResolve(_scope) && this.defaut.fullResolve(_scope);
 return result;
 }
}
```

Typage

a- traitements nécessaire pour gérer le typage

1. Typage de l'expression
2. Iterate over the list of all cases

Check that value same type as expression

Recursive call on block

3. Recursive call on block by default

b-

```
boolean checkType() {
 boolean result = true;
 Type type = this.expression.getType();
 for(Case c : this.cases){
 result = result && c.getValue.compareWith(type);
 result = result && c.getBlock.checkType();
 }
 result = result && this.defaut.checkType();
 return result;
}
```

Génération du code

a- Traitements necessaire pour gérer la génération de code

1. Reserve spaces for results (PUSH size_result)
2. Generate code for expression
3. assign the result to the variable (STORE (size_result) @_result)
4. Iterate over the list of all cases
   - Read the value of result (LOAD (size_result) @_result)
   - Load value associated with the case c (LOADL case_value)
   - Comparer (SUBR IEq)
   - If the judgment condition is not met, jump to the else branch (JUMPIF (0) case_else)
   - Generate the code of the block associated with the case c
   - Jump to the fin branch (JUMP case_fin)
   - Generate case_else label
5. Generate code for defaut
6. Generate case_fin label

Conditional

```
public class Conditional implements Instruction {
	protected Expression condition;
	protected Block thenBranch;
	protected Block elseBranch;
	public Conditional(Expression _condition, Block _then, Block _else) {
	this.condition = _condition;
	this.thenBranch = _then;
	this.elseBranch = _else;
	}
	public Conditional(Expression _condition, Block _then) {
	this.condition = _condition;
	this.thenBranch = _then;
	this.elseBranch = null;
	}
	@Override
	public String toString() {
		return "if (" + this.condition + " )" + this.thenBranch + ((this.elseBranch != null)?(" else
		" + this.elseBranch):"");
	}
	@Override
	public boolean collectAndBackwardResolve(HierarchicalScope<Declaration> _scope) {
		if (elseBranch == null) {
			return this.condition.collectAndBackwardResolve(_scope) &&
		this.thenBranch.collect(_scope);
		} else {
			return this.condition.collectAndBackwardResolve(_scope) &&
		this.thenBranch.collect(_scope) && this.elseBranch.collect(_scope);
		}
	}
	@Override
	public boolean fullResolve(HierarchicalScope<Declaration> _scope) {
		if (elseBranch == null) {
			return this.condition.fullResolve(_scope) && this.thenBranch.resolve(_scope);
		} else {
			return  this.condition.fullResolve(_scope) && this.thenBranch.resolve(_scope)
		&& this.elseBranch.resolve(_scope);
		}
	}
	@Override
	public boolean checkType() {
		boolean result = this.condition.getType().compatibleWith(AtomicType.BooleanType);
		if (elseBranch == null) {
			result = result && this.thenBranch.checkType() ;
		} else {
			result = result && this.thenBranch.checkType() &&
		this.elseBranch.checkType();
		}
		return result;
	}
	@Override
	public int allocateMemory(Register _register, int _offset) {
		this.thenBranch.allocateMemory(_register, _offset);
		if (this.elseBranch != null) {
			this.elseBranch.allocateMemory(_register, _offset);
		}
		return 0;
	}
	@Override
	public Fragment getCode(TAMFactory _factory) {
		Fragment _result = _factory.createFragment();
		int id = _factory.createLabelNumber();
		_result.append(this.condition.getCode(_factory));
	if (this.elseBranch == null) {
		_result.add(_factory.createJumpIf("endif" + id, 0));
		_result.append(this.thenBranch.getCode(_factory ));
	} else {
		_result.add(_factory.createJumpIf("else" + id, 0));
		_result.append(this.thenBranch.getCode(_factory));
		_result.add(_factory.createJump("endif" + id));
		_result.addSuffix("else" + id);
		_result.append(this.elseBranch.getCode(_factory));
	}
	_result.addSuffix("endif" + id);
	return _result;
}
}
```