Questions de Cours

- **Question 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.

- **Question 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

- **Question 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

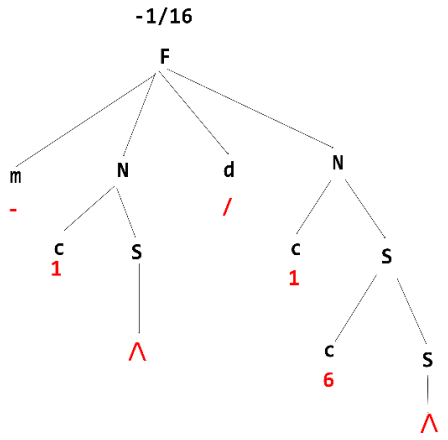
Les attributs sont divisés en deux groupes : les attributs synthétisés et les attributs hérités. Les attributs synthétisés sont le résultat des règles d'évaluation des attributs ; ils peuvent aussi utiliser les valeurs d'attributs hérités. Les attributs hérités sont passés vers les feuilles à partir 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

- **Question 4 (simple)**

{i -> 1} |- (function j -> i + j) (2) => 3

- Etapes 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

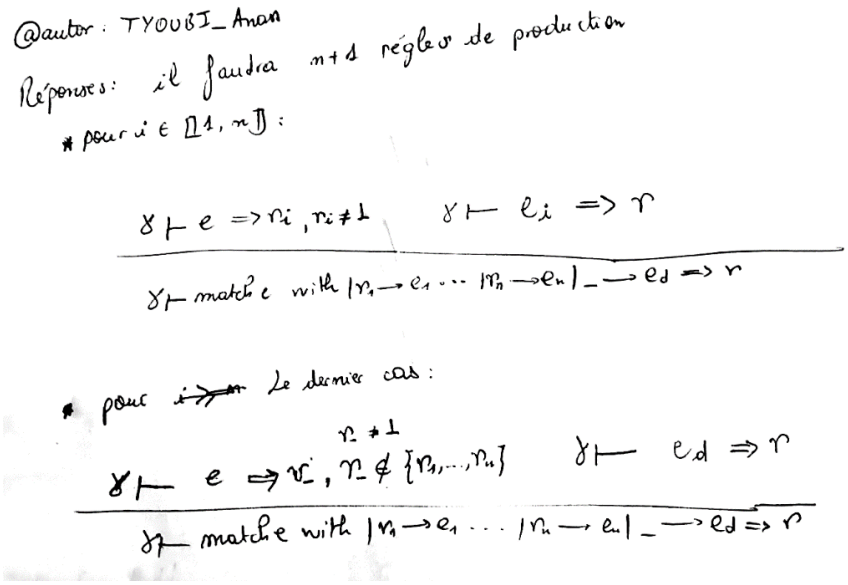


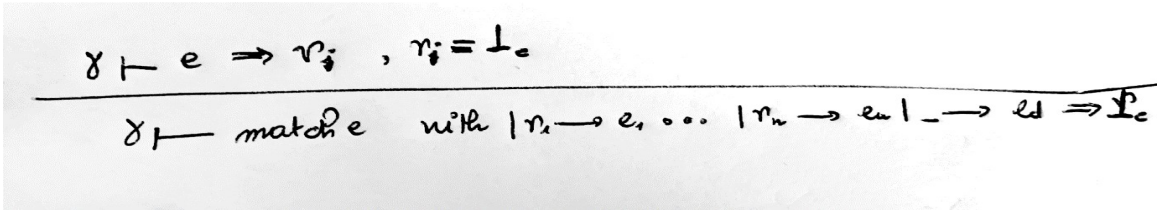
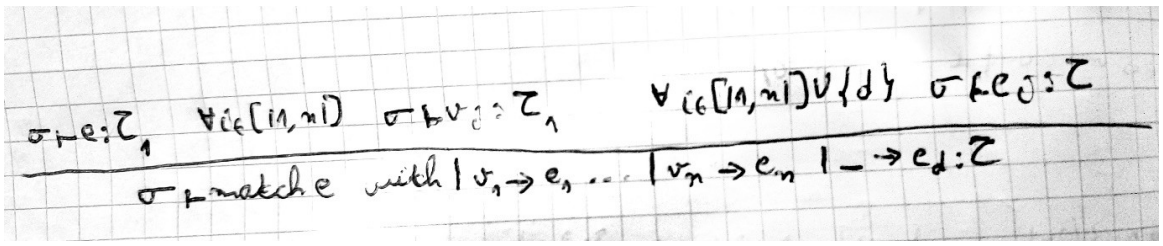
Sémantique attribuée

- F : valeur : entier
- V : valeur : entier
- S : valeur : entier
  profondeur : entier
- `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

- 

- 
- 

d)

```
rule match selection e1 ... en default env =
    let etype = (type_of_expr selection env) in
            let etype1 = (type_of_expr e1 env) in
```

```
                let etype2 = (type_of_expr e1 env) in
                        ....
                        let etypen = (type_of_expr en env) in
                                let _bool,_type = unify etype etype1 ... etype2 in
                                    if _bool then
                                        _type
                                    else
                                        ErrorType
```

e)

```
rule match selection e1 ... en default env =
        let eval1 = (value_of_expression selection env)
        in
        match eval1 with
                |(ErrorValue _) as result -> result
                | _ as eval2 -> match eval2 with
                        | v1 -> (value_of_expression e1 env)
                        | ......
                        | vn -> (value_of_expression en env)
                        | _ -> (value_of_expression default env)
```

Instruction de choix en Bloc

```
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]
```

```
// i + 1
LOADL 1
SUBR IAdd

LOADL 0
SUBR IEq
JUMPIF (0) case_1
SUBR COut

case_1:
POP (0) (1)
LOADL 1
SUBR IEq
JUMPIF (0) case_5
SUBR COut

case_5:
POP (0) (1)
LOADL 5
SUBR IEq
JUMPIF (0) case_default
SUBR COut

case_default:
POP (0) (1)
LOADL -1
SUBR COut
```

```
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;
    }
    @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;
    }
}
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;
		}
}

block {
   int fact( int n) {
      if (n == 0) {
         return 1;
      } else {
         return n * fact(n - 1);
      }
   }
   print fact(5);
}
LOADL 5
CALL (SB) begin:fact
SUBR IOut
LOADL 10
SUBR COut
POP (0) 0
```

```
HALT
begin:fact
LOAD (1) -1[LB]
LOADL 0
SUBR IEq
JUMPIF (0) else1
LOADL 1
RETURN (1) 1
JUMP endif1
else1
LOAD (1) -1[LB]
LOAD (1) -1[LB]
LOADL 1
SUBR ISub
CALL (SB) begin:fact
SUBR IMul
RETURN (1) 1
endif1
end:fact

block {

  int pgcd( <int,int> c ) {
    int a = (fst c);
    int b = snd c;
    while (a * b != test) {
      if ( a > b ) {
                  int na = a-b;
                  a = na;
      } else {
                  int nb = b-a;
                  b = nb;
      }
    }
    int res = a;
    if (res == test) {
      res = b;
```

```
    }
    return res;
  }
  const int test = 0;
  print pgcd( <47, 53> );
}

LOADL 0
LOADL 47
LOADL 53
CALL (SB) begin:pgcd
SUBR IOut
LOADL 10
SUBR COut
POP (0) 1
HALT
begin:pgcd
PUSH 1
LOAD (2) -2[LB]
POP (0) 1
STORE (1) 3[LB]
PUSH 1
LOAD (2) -2[LB]
POP (1) 1
STORE (1) 4[LB]
while1
LOAD (1) 3[LB]
LOAD (1) 4[LB]
SUBR IMul
LOADL 0
SUBR INeq
JUMPIF (0) endwhile1
LOAD (1) 3[LB]
LOAD (1) 4[LB]
SUBR IGtr
JUMPIF (0) else2
PUSH 1
```

```
LOAD (1) 3[LB]
LOAD (1) 4[LB]
SUBR ISub
STORE (1) 5[LB]
LOAD (1) 5[LB]
STORE (1) 3[LB]
JUMP endif2
else2
PUSH 1
LOAD (1) 4[LB]
LOAD (1) 3[LB]
SUBR ISub
STORE (1) 5[LB]
LOAD (1) 5[LB]
STORE (1) 4[LB]
endif2
JUMP while1
endwhile1
PUSH 1
LOAD (1) 3[LB]
STORE (1) 5[LB]
LOAD (1) 5[LB]
LOADL 0
SUBR IEq
JUMPIF (0) endif3
LOAD (1) 4[LB]
STORE (1) 5[LB]
endif3
LOAD (1) 5[LB]
RETURN (1) 1
end:pgcd
```