# Questions du projet de compilation

Thomas-Alexandre Moreau & Nicolas Cornu–Laporte

## 1 Questions - Lexing

*Dans votre document, vous préciserez les difficultés rencontrées si votre lexeur n'accepte pas tout ce qui est décrit ici. Vous expliquerez la solution mise en œuvre pour reconnaître les commentaires sur plusieurs lignes tout en maintenant le compteur de ligne du buffer cohérent avec le fichier analysé.*

Le lexeur accepte bien tout ce qui est décrit (testé avec tous les programmes mis à disposition). Pour ce qui est des commentaires sur plusieurs lignes, nous avons utilisé le même procédé que lors des TPs (accessible notamment sur le TP5).

## 2 Questions - Parsing

*Dans votre document, vous préciserez les difficultés rencontrées si votre parseur n'accepte pas tout ce qui est décrit ici. Vous pouvez bien évidemment jouer avec menhir pour répondre à ces questions.*

Le parseur n'a aucun problème sur aucun des programmes fournis, ni même de conflits. Cependant il y a pour l'instant deux problèmes au niveau de l'AST pour les programmes *renaming.pix* et *inner_overwrite.pix*, les deux provoquant l'erreur suivante : *inconsistant operands*. Cette erreur correspond à priori à une erreur de types sur les opérations binaires, mais après vérification lors du parsing, il n'y a pas de problèmes de types apparents.

### 2.1

*On considère la séquence suivante : **If** #**expr**# **If** #**expr**# #**stmt**# **Else** #**stmt**# On considère pour cette question que #**expr**# et #**stmt**# sont des terminaux (i.e., on ne cherchera pas à les «étendre»).*

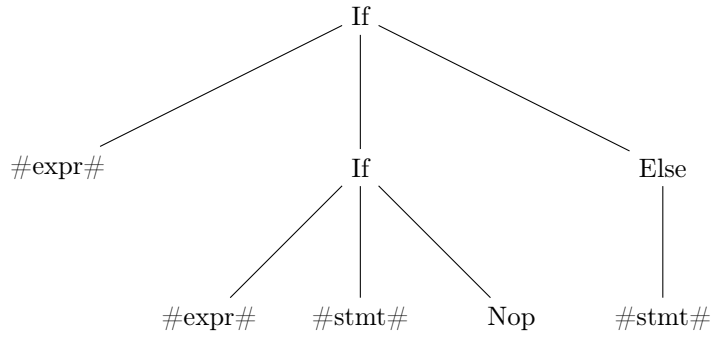
#### 2.1.1

*Donnez les deux arbres de dérivations possible de cette séquence dans la grammaire décrite plus haut.*

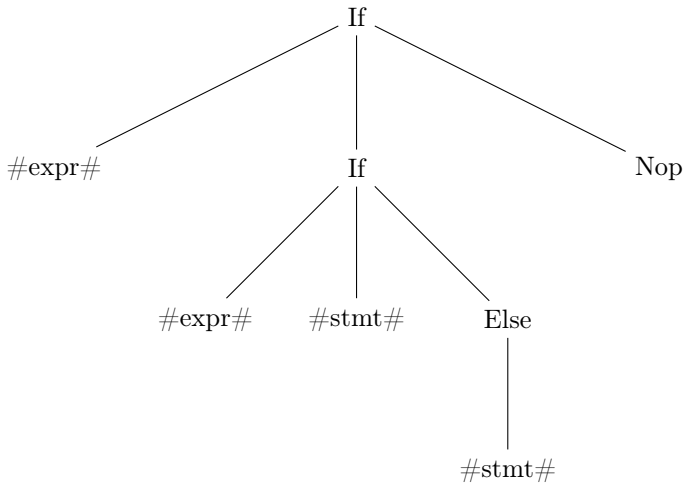
La différence entre les deux arbres se joue au niveau du **Else**. Etant donnée la sémantique du **If** suivante : **If** #**expr**# #**stmt**# ou bien **If** #**expr**# #**stmt1**# **Else** #**stmt2**#, on peut soit reduce le **Else** de l'exemple avec le premier **If** ou le second.

Exemple (on se servira de couleurs pour représenter les "blocs" pour le premier **If**, rouge pour représenter #**stmt1**# ou simplement #**stmt**# et bleu pour représenter #**stmt2**# s'il existe) :

- **If** #**expr**# **If** #**expr**# #**stmt**# **Else** #**stmt**#
- **If** #**expr**# **If** #**expr**# #**stmt**# **Else** #**stmt**#



Arbre de dérivation 1



Arbre de dérivation 2

**2.1.2**

*Donnez l'état de l'automate LR0 où apparaît le conflit qui montre l'existence de ces deux arbres.*

Tout d'abord, un conflit peut se produire entre autre lorsqu'une action de réduction et une action de décalage (shift reduce) sont possibles pour le même symbole à partir du même état. Dans ce cas, pour l'automate LR0, il y a un conflit au niveau de l'état 134

```
** Conflict (shift/reduce) in state 134.
** Token involved: ELSE
** This state is reached from main after reading:

IF L_PAR expression R_PAR IF L_PAR expression R_PAR statement

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin to differ.)

main
program EOF
statement
(?)

** In state 134, looking ahead at ELSE, shifting is permitted
```

```
** because of the following sub-derivation:

IF L_PAR expression R_PAR statement ELSE statement
                            IF L_PAR expression R_PAR statement . ELSE statement

** In state 134, looking ahead at ELSE, reducing production
** statement -> IF L_PAR expression R_PAR statement
** is permitted because of the following sub-derivation:

IF L_PAR expression R_PAR statement ELSE statement // lookahead token appears
                            IF L_PAR expression R_PAR statement .
```

Comme on peut le voir, il y a un conflit shift-reduce sur cet état.

#### 2.1.3

*Quelle annotation permet d'obtenir l'arbre cohérent avec la priorité décrite dans ce document ?*

Pour information : Le *Else* est facultatif dans le If-Else. Un *Else* se rattache toujours au $If$ non terminé le plus proche.
Pour obtenir l'arbre cohérent avec la priorité, il faut utiliser *%right ELSE* pour donner la priorité au shift, et ainsi avec la séquence précédente : **If** #**expr**# **If** #**expr**# #**stmt**# **Else** #**stmt**#, le *Else* sera bien associé au second $If$. Cependant pour que cela fonctionne réellement nous avons dû rajouter d'autres annotations :

```
statement:
| IF L_PAR e = expression R_PAR s = statement ifife = ife
{ IfThenElse (e, s, ifife, Annotation.create $loc) }

%inline ife:
| ELSE s = statement { s }
| { Nop }
%prec ELSE
```

Ici, nous avons uniquement représenté ce qu'il implique le $If$ et le $ELse$. En combinant la règle inlinée, l'annotation %prec ELSE et l'annotation %right ELSE, tous les conflits ont disparu et le Else s'associait bien aux If le plus proche.

#### 2.1.4

*Pouvez-vous via une annotation obtenir le comportement inverse ? Pourquoi ?*

Par défaut, l'association se fait à droite, mais il ne semble pas possible de pouvoir inverser ce comportement.

### 2.2

*Choisissez un conflit shift-reduce possible dans votre grammaire sans annotation, et expliquez quelles annotations de priorité vous avez mis pour le résoudre, et son effet sur les arbres acceptés. Vous illustrerez un exemple où ce conflit pourrait arriver et les deux arbres mis en jeu via une séquence de tokens.*

Si on reprend le conflit shift-reduce du $If\ ELse$. Nous constatons bien un conflit dans notre grammaire sans annotation. Et pour eviter des différences sur les arbres accaptés, nous devons rajoute une combinaison *%right ELSE*

# 3 Questions - Renommage de variable

*Dans votre document, vous préciserez les difficultés rencontrées si votre renommage n'effectue pas tout ce qui est décrit ici.*

Aucun problème rencontré. De plus, après implémentation du renommage, tous les programmes fournis s'exécutent sans problèmes (notamment les deux cités précédemment qui ne s'exécutaient pas).

## 3.1

*Pourquoi n'est-il pas gênant que dans deux blocs disjoints (pas l'un dans l'autre) un même nom soit utilisé pour des variables locales à ces blocs ?*

Imaginons qu'une variable "x" soit utilisée dans deux blocs disjoints, par exemple :

```
$<
    Int : x;
    Set(x, 10);
    Print(x);
>$
$<
    Int : x;
    Set(x, 20);
    Print(x);
>$
```

Le premier **Print** affichera 10, et le second affichera 20, bien que la variable ait le même nom dans les deux blocs, car chaque variable "x" des deux blocs est en fait totalement indépendante : la première est **locale** au premier bloc et la seconde est **locale** au deuxième bloc. Ainsi quelque soit les modifications faites sur la variable "x" du premier bloc, cela n'aura absolument aucun impact dans le second.

## 3.2

*Dans le programme suivant (il s'agit du programme renaming.pix, qui n'a pas d'intérêt particulier autre que pour cet exercice), indiquez comment le renommage des variables sera effectué.*

```
<
    Int : x;
    Real : y
>
$<
    Set(x,2*x);
    Real : x;
    Set(x,2.0*y);
    Int : y;
    Set(y,Floor(x));
    $<
        Int : x;
        Set(x,2*y);
        Int : y;
        Set(y,2*x);
    >$;
    Set(y,2*y);
    $<
```

```
        Coord : x;
        Set(x,Coord(y,y));
        Color : y;
        Set(y,Color(x.X,x.X,x.X));
        Draw(Pixel(x,y));
    >$;
    Set(x,2.2*x);
>$
```

Après analyse, cela donne :

```
<
  Int : x;
  Real : y
  >
$<
    Set(x, (2*x));
    Real : x#1;
    Set(x#1, (2.*y));
    Int : y#1;
    Set(y#1, Floor(x#1));
    $<
        Int : x#2;
        Set(x#2, (2*y#1));
        Int : y#2;
        Set(y#2, (2*x#2));

    >$;
    Set(y#1, (2*y#1));
    $<
        Coord : x#2;
        Set(x#2, Coord(y#1, y#1));
        Color : y#2;
        Set(y#2, Color(x#2.X, x#2.X, x#2.X));
        Draw_pixel(Pixel(x#2, y#2));

    >$;
    Set(x#1, (2.2*x#1));

>$
```

# 4 Questions - Analyse de types

## 4.1

*Pourquoi a-t'on besoin de Type_generic pour la liste vide ?*

Une liste vide n'ayant aucun élément d'un type particulier, il faut pouvoir lui affecter un type sans pour autant poser des problèmes plus tard (exemple : partir du principe que c'est une liste de *Int* alors que finalement cette liste est utilisée avec des *Real*. Donc avoir un type générique pour traiter le cas de la liste vide est une bonne solution.

## 4.2

*Pourquoi doit-on réaliser des copies des environnements avant de vérifier la cohérence des types à l'intérieur des blocs ?*

La vérification de types fonctionne uniquement par effets de bord (sauf pour les expressions). Donc il est nécessaire de faire des copies des environnements initiaux afin d'avoir des environnements réservés à la vérification de types pour que celle-ci n'ait pas d'effets directs sur ces environnements de base. La vérification de type est comme son nom l'indique une vérification, elle ne doit pas avoir d'impact direct sur le programme.

## 4.3

*Donnez un tableau similaire à celui des opérateurs unaires qui précise le typage des champs (fields).*

| AST | Type compatible | Type de sortie |
|---|---|---|
| *Color_field* | *Type_pixel* | *Type_color* |
| *Coord_field* | *Type_pixel* | *Type_coord* |
| *X_field* | *Type_coord* | *Type_int* |
| *Y_field* | *Type_coord* | *Type_int* |
| *Red_field* | *Type_color* | *Type_int* |
| *Green_field* | *Type_color* | *Type_int* |
| *Blue_field* | *Type_color* | *Type_int* |

## 4.4

*Expliquez quelles sont les deux erreurs qu'on peut détecter lorsqu'on traite un statement* $For(x, start, last, step, body)$ *lors de l'analyse de type*

Si les types de *start*, *last* et *step* sont différents, on a une erreur :
"For loop with inconsistent types $t\_start$, $t\_last$ and $t\_step$"

Il y aurait effectivement un problème si l'on partait par exemple de i = 0, pour aller à i = 10.2.

Si le type de *start* n'est ni *Int* ni *Real*, on a une erreur :
"For loop on $t\_start$ instead of Int or Real"

Ici, il est évident que l'on a besoin d'itérer sur des entiers ou des réels.

Enfin, puisque le type de *start* est nécessairement *Int* ou *Real*, et que les types de *start*, *last* et *step* doivent être les mêmes, alors les types de *last* et *step* doivent être *Int* ou *Real*. Tous les cas d'erreurs possibles sont ainsi bien traités.

# 5 Questions - Simplification

*Dans votre document, vous préciserez les difficultés rencontrées si votre simplificateur n'effectue pas tout ce qui est décrit ici.*

## 5.1

*Pourquoi peut-on éliminer les For dans le cas décrit ci-dessus ?*

Si on a un *For* qui a par exemple, comme valeur de départ 5 et valeur d'arrivée 3, on entrera jamais dans la boucle puisque $5 > 3$, donc remplacer par un bloc vide aura le même effet puisque aucune instruction du bloc *For* ne sera exécuté.

## 5.2

*Pourquoi ne simplifie t'on pas les nœuds de la forme* $Real\_of\_int(Floor(n, a1), a2)$ *?*

Prenons n = 10.2 par exemple, si l'on fait $Floor(n)$ on obtient 10, ensuite on fait $Real_of_int(n)$ ce qui donnera 10.0.

Comme on peut le voir, 10.0 != 10.2, tandis que dans l'autre sens si n = 10 -> $Real\_of\_int(n)$ = 10. -> $Floor(n)$ = 10 == n (donc ce cas est bien simplifiable).

Le résultat est donc bien différent suivant l'ordre d'exécution de $Real\_of\_int$ et $Floor$.

# 6 Questions - Simplification++

Tout d'abord, on peut rajouter un cas de simplification au *For* : Si on sait à l'avance qu'une boucle *For* ne fera qu'un seul tour de boucle, on pourra simplement remplacer ce *For* par un *Block* simple. Idem pour *Foreach*.

Le *For* est à priori bien implémenté comme il faut, ainsi que le *Foreach*. Après plusieurs tests cela fonctionnait comme convenu.
Cependant si le *For* ou *Foreach* utilisent des arguments, dans ce cas ça ne transforme pas en *Block* simple.

Deux programmes ont été créés pour tester la fonctionnalité :
- *programs/custom_examples/for_extension*
- *programs/custom_examples/foreach_extension*

Concernant la partie sur les opérations arithmétiques, nous ne l'avons pas implémenté, mais nous pouvons expliquer rapidement le procédé. Par exemple si nous avons l'instruction suivante : $Set(x, 3 * 4)$ il faudrait à la place remplacer cela par $Set(x, 2 * 12$ ou bien $Set(x, 10 + 2$. Il y aurait évidemment plein de choix possibles.

Quant à son implémentation, il faudrait certainement calculer les résultats en amont pour ensuite les décomposer en addition, soustraction, ...

Cependant certains cas pourraient être plus triviaux comme par exemple 3 * 3, que l'on pourrait transformer en 3 + 3 + 3 assez facilement (simple récursion).