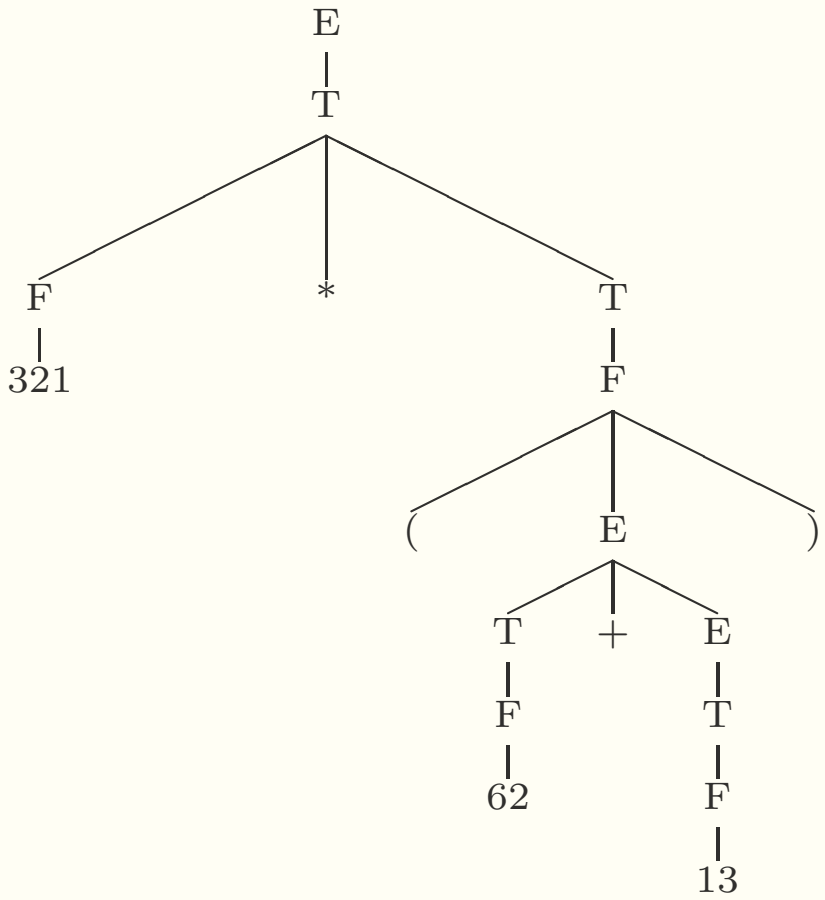
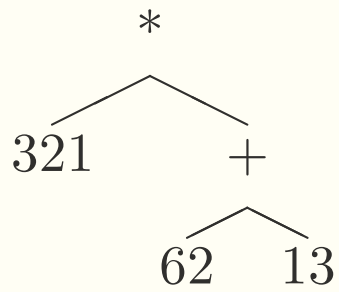
Arbre de dérivation v/s arbre abstrait

- L'arbre de dérivation produit par l'analyse syntaxique possède de nombreux nœuds superflus, qui ne véhiculent pas d'information.
- De plus, la mise au point d'une grammaire (élimination de l'ambiguïté, élimination de la récursivité à gauche, factorisation) nécessite souvent l'introduction de règles dont le seul but est de simplifier l'analyse syntaxique.
- Un arbre abstrait constitue une interface plus naturelle entre l'analyse syntaxique et l'analyse sémantique, elle ne garde de la structure syntaxique que les parties nécessaires à l'analyse sémantique et à la production de code.
- L'arbre abstrait est construit lors de l'analyse syntaxique, en associant à toute règle de grammaire une action sémantique.

Arbre de dérivation v/s arbre abstrait





Traduction dirigée par la syntaxe

La traduction dirigée par la syntaxe est réalisée en attachant des actions sémantiques aux règles de la grammaire.
Elle repose sur deux concepts :

- Les attributs. Un attribut est une quantité quelconque associée à une construction du langage de programmation.
- Exemples :
  - le type d'une expression
  - la valeur d'une expression
  - le nombre d'instructions dans le code généré
- Les constructions étant représentées par les symboles de la grammaire, on associe les attributs à ces derniers.
- Notations : A.t est l'attribut t associé au symbole A.

Traduction dirigée par la syntaxe

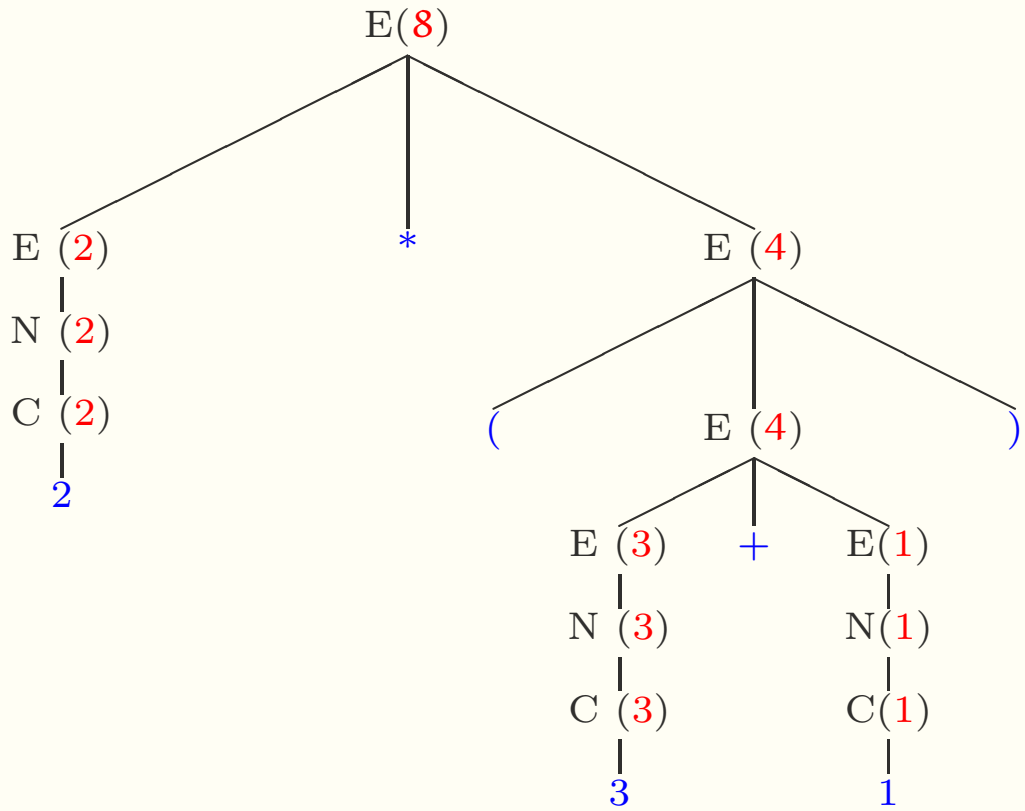
- Les schémas de traduction sont une notation permettant d'attacher des fragments de programme aux règles de la grammaire.
- Les fragments sont exécutés quand la production est utilisée lors de l'analyse syntaxique.
- Le résultat combiné des exécutions de tous ces fragments, dans l'ordre induit par l'analyse syntaxique, produit la traduction du programme auquel ce processus est appliqué.

Exemple 1

règle			action sémantique		
E	$\rightarrow$	E + E	E.t	=	$E_1.t + E_2.t$
E	$\rightarrow$	E * E	E.t	=	$E_1.t * E_2.t$
E	$\rightarrow$	(E)	E.t	=	$E_1.t$
E	$\rightarrow$	N	E.t	=	N.t
N	$\rightarrow$	C N	N.t	=	$10 * C.t + N_2.t$
N	$\rightarrow$	C	N.t	=	C.t
C	$\rightarrow$	0	C.t	=	0
C	$\rightarrow$	1	C.t	=	1
C	$\rightarrow$	2	C.t	=	2
	...			...	
C	$\rightarrow$	9	C.t	=	9

Attention : E_1 et E_2 sont des instances du non terminal E et non des symboles différents. Il est nécessaire de les distinguer afin de distinguer les traductions qui leurs sont associées.

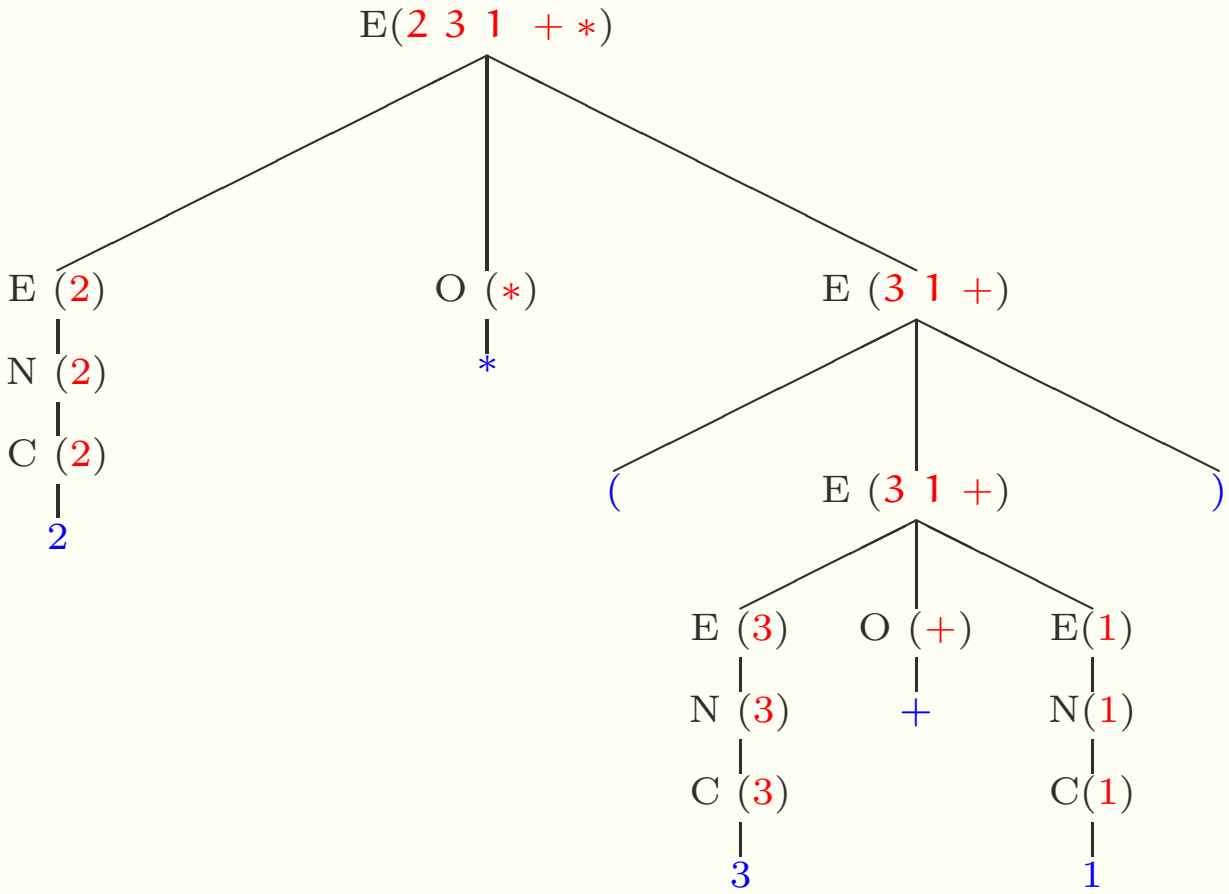
Exemple 1



Exemple 2

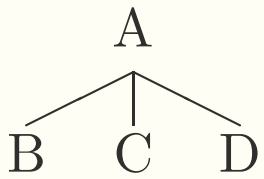
règle			action sémantique		
E	$\rightarrow$	E O E	E.t	=	E_1.t $\|\|$ E_2.t $\|\|$ O.t
E	$\rightarrow$	(E)	E.t	=	E_1.t
E	$\rightarrow$	N	E.t	=	N.t
O	$\rightarrow$	+	O.t	=	+
O	$\rightarrow$	−	O.t	=	−
N	$\rightarrow$	C N	N.t	=	C.t $\|\|$ N_2.t
N	$\rightarrow$	C	N.t	=	C.t
C	$\rightarrow$	0	C.t	=	0
C	$\rightarrow$	1	C.t	=	1
C	$\rightarrow$	2	C.t	=	2
	...			...	
C	$\rightarrow$	9	C.t	=	9

Exemple 2



Attributs synthétisés

- Un attribut est dit synthétisé si sa valeur au niveau d'un nœud A d'un arbre d'analyse est déterminée par les valeurs de cet attribut au niveau des fils de A et de A lui même.



$$A.t = f(B.t, C.t, D.t)$$

- Les attributs synthétisés peuvent être évalués au cours d'un parcours ascendant de l'arbre de dérivation.
- Un tel parcours peut être effectué simultanément à la construction de l'arbre (lors de l'analyse syntaxique).
- Dans l'exemple, B.t, C.t et D.t doivent être calculés avant de calculer A.t

Construction de l'arbre abstrait à l'aide de schémas de traduction

règle			action sémantique		
instSi	$\rightarrow$	SI exp ALORS inst	instSi.n	=	noeud(SI, exp.n, inst.n)
exp0	$\rightarrow$	conj OU exp1	exp0.n	=	noeud(OU, conj.n, exp1.n)
exp	$\rightarrow$	conj	exp.n	=	conj.n
LInst0	$\rightarrow$	inst LInst1	LInst0.n	=	noeud(LI, inst.n, LInst1.n)
LInst0	$\rightarrow$	ε	LInst0.n	=	NULL

Mise en œuvre en C

```
void instructionSi(void)
{
  if(uc != SI)
    erreur("SI attendu");
  uc = yylex();
  expression();
  if(uc != ALORS)
    erreur("ALORS attendu");
  uc = yylex();
  instruction();
  return;
}
```

Mise en œuvre en C

```
void instructionSi(void)
{
  if(uc != SI)
    erreur("SI attendu");
  uc = yylex();
  expression();
  if(uc != ALORS)
    erreur("ALORS attendu");
  uc = yylex();
  instruction();
  return;
}
```

```
n_instr *instructionSi(void)
{
  n_instr *S0 = NULL;
  n_exp   *S2 = NULL;
  n_instr *S4 = NULL;

  if(uc != SI)
    erreur("SI attendu");
  uc = yylex();
  S2 = expression();
  if(uc != ALORS)
    erreur("ALORS attendu");
  uc = yylex();
  S4 = instruction();
  S0 = cree_n_instr_si(S2, S4);
  return S0;
}
```

Structure n_instr

```
typedef struct {
    n_exp *test;
    struct n_instr_ *alors;
    struct n_instr_ *sinon;
} n_instr_si;
```

Problème : un type de nœud différent par type d'instruction

```
typedef struct {
    n_exp *test;
    struct n_instr_ *faire;
} n_instr_tantque;
```

Il faut prévoir tous les cas dans le code !

Polymorphisme

Autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

```
enum {inst_si, inst_tantque} type_inst;

typedef struct {
    type_inst type;
    /* champs specifiques au type inst_si */
    n_exp *test;
    struct n_instr_ *alors;
    struct n_instr_ *sinon;
    /* champs specifiques au type inst_tantque */
    n_exp *test;
    struct n_instr_ *faire;
} n_instr;
```

pas très économique, seuls quelques champs seront utilisés dans chaque cas !

Polymorphisme

On peut gagner un peu de place en factorisant

```
enum {inst_si, inst_tantque} type_inst;

typedef struct {
    type_inst type;
    n_exp *test;
    /* champs specifiques au type inst_si */
    struct n_instr_ *alors;
    struct n_instr_ *sinon;
    /* champs specifiques au type inst_tantque */
    struct n_instr_ *faire;
} n_instr;
```

Mais il y aura toujours de la perte !

union

"Tandis que les champs des structures se suivent sans se chevaucher, les champs d'une union commencent tous au même endroit et, donc, se superposent. Ainsi, une variable de type union peut contenir, à des moments différents, des objets de types et de tailles différents. La taille d'une union est celle du plus volumineux de ses champs."
C : langage, bibliothèque, applications Henri Garreta, p.94

```
typedef union {
  double flottant;
  int    entier;
  char   caractere;
} scalaire;

scalaire x;

x.flottant = 1.23;
x.entier = 123;
x.caractere = 'c';
```

Le type n_instr

```
typedef struct n_instr_ n_instr;

struct n_instr_ {
  enum {affecteInst, siInst, tantqueInst, appelInst,
        retourInst, ecrireInst, videInst, blocInst} type;
  union{
    struct{n_exp *test; struct n_instr_ *alors;
           struct n_instr_ *sinon;} si_;
    struct{n_exp *test;
           struct n_instr_ *faire;} tantque_;
    n_appel *appel;
    struct{n_var *var; n_exp *exp;} affecte_;
    struct{n_exp *expression;} retour_;
    struct{n_exp *expression;} ecrire_;
    n_l_instr *liste;
  }u;
};
```

Un constructeur par type d'instruction

```
n_instr *cree_n_instr_si(n_exp *test,
                         n_instr *alors, n_instr *sinon)
{
  n_instr *n = malloc(sizeof(n_instr));
  n->type = siInst;
  n->u.si_.test = test;
  n->u.si_.alors = alors;
  n->u.si_.sinon = sinon;
  return n;
}
```