Traduction et Sémantique
Traduction dirigée par la syntaxe

Luigi Santocanale
LIF, Université de Provence
Marseille, FRANCE

19 mars 2010

Plan

Plan

Définitions dirigées par la syntaxe

Grammaire attribuée :
toute production possède un ensemble de règles sémantiques :

Production	Règle sémantique
⋮	
$A \to X_1 \dots X_n$	a_1
	a_2
	⋮
	a_k
⋮	

Chaque règle sémantique (ou action) a_i :
- calcule un attribut d'un symbole parmi $A, X_1, \dots, X_n$, ou ...
- ...produit un effet de bord.

Grammaire attribuée pour une calculette

Production	Règle sémantique
$L \to E\ \backslash\mathtt{n}$	*Imprimer*($E.val$)
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val \cdot F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathtt{chiffre}$	$F.val := \mathtt{chiffre}.vallex$

Attribution de types dans une déclaration

Production	Règle sémantique
$D \to T\ L$	$L.type := T.type$
$T \to \mathtt{entier}$	$T.type :=$ *entier*
$T \to \mathtt{reel}$	$T.type :=$ *reel*
$L \to L_1\ ,\ \mathtt{id}$	$L_1.type := L.type$ *AjouterType*($\mathtt{id}.entree, L.type$)
$L \to \mathtt{id}$	*AjouterType*($\mathtt{id}.entree, L.type$)

Formellement

Une grammaire attribuée consiste de :

- un grammaire $\mathscr{G} = \langle V, \Sigma, S, P \rangle$
- pour tout $X \in V$, une collection d'attributs de X :

$$\{ X.i \mid i \in I_X \}$$

- pour toute production $A \to X_1, \ldots, X_n$,
  une collection de règles a_i de la forme

$$b := f(c_1, c_2, \ldots, c_k)$$

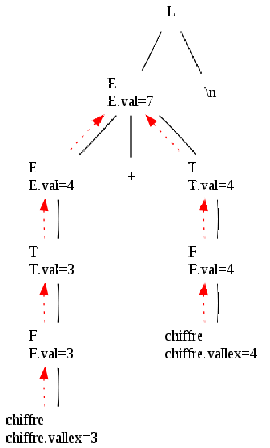
où $b, c_1, \ldots, c_k$ sont des attributs de $A, X_1, \ldots, X_n$.

Attributs synthétisés : intuition

Le flot de l'information procède des fils vers les parents,
et de droite à gauche du symbole $\to$ de la production :

Exemple, la ligne

$3 + 4\backslash\mathtt{n}$



Attributs synthétisés

Un attribut b est synthétisé si chaque fois que

Production	Règle sémantique
⋮	
$A \to X_1 \dots X_n$	⋮
	$b := f(c_1, \dots, c_k)$
	⋮
⋮	

b est un attribut de A,
c'est-à-dire il est de la forme $A.i$.

L'attribut E.val de la calculette est synthétisé

Production	Règle sémantique
$L \to E \backslash \mathtt{n}$	$\mathit{Imprimer}(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val \cdot F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathtt{chiffre}$	$F.val := \mathtt{chiffre}.vallex$

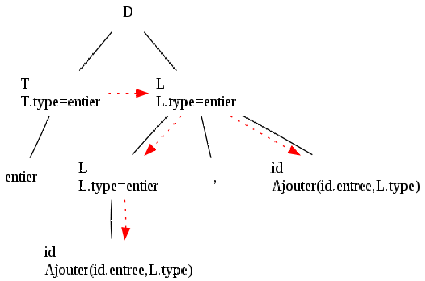
Tous les attributs de la calculette sont synthétisés.

Attributs hérités

Le flot d'information procède des
parents vers les fils, entre frères,
(et de gauche à droite d'une production.)

Exemple,
la déclaration

```
entier i,j
```



Attributs hérités

Un attribut b est hérité si chaque fois que

Production	Règle sémantique
⋮	
$A \to X_1 \dots X_n$	⋮
	$b := f(c_1, \dots, c_k)$
	⋮
⋮	

b est un attribut de X_j, pour quelque $j = 1, \dots, n$,
c'est-à-dire il est de la forme $X_j.i$.

L'attribut L.type est hérité.

Production	Règle sémantique
$D \rightarrow T\ L$	$L.type := T.type$
$T \rightarrow$ `entier`	$T.type := entier$
$T \rightarrow$ `reel`	$T.type := reel$
$L \rightarrow L_1$, `id`	$L_1.type := L.type$ $AjouterType(\texttt{id}.entree, L.type)$
$L \rightarrow$ `id`	$AjouterType(\texttt{id}.entree, L.type)$

$L.type$ est hérité, $T.type$ est synthétisé.

Implantation ascendante des définitions S-attribués

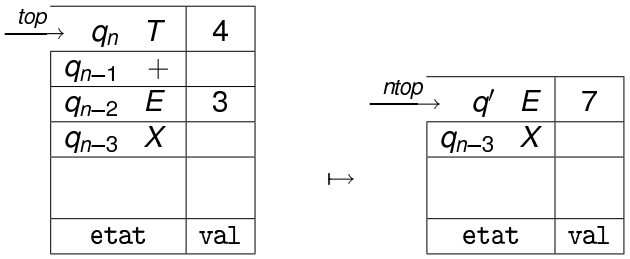
Grammaire *S*-attribués :
tous les attributs sont synthétisés.

Implantation

- On étend un analyseur ascendante.
- On place les attributs sur la pile.
- On calcul les attributs du père, avant réduction.

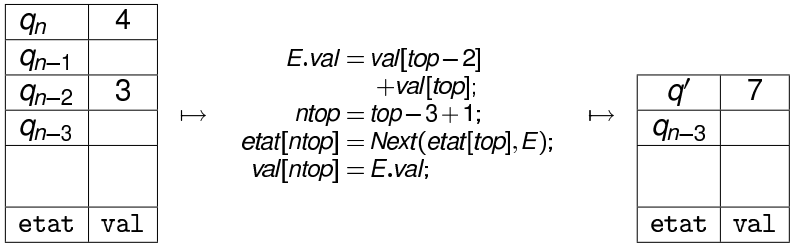
Exemple

Production	Règle sémantique
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$



Exemple (cont.)

Production	Règle sémantique
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$



Production	Règle sémantique
$E \rightarrow E_1 + T$	$val[ntop] := val[top-2] + val[top]$

Construction de la syntaxe abstraite pour le langage `lse`

La syntaxe abstraite en `tic` (fichier lse_syntabstr.tic) :

```
type op = Plus ;

type e_expr = Id of string
     | Const of int
     | Op of op * e_expr * e_expr ;

type b_expr = Expr of e_expr
     | Ite of e_expr * b_expr * b_expr
     | Let of string * e_expr * b_expr ;
```


Les prototypes engendrés (fichier lse_syntabstr.tic.h) :

```
extern Let *make_Let(char *,e_expr *,b_expr *);
extern Ite *make_Ite(e_expr *,b_expr *,b_expr *);
extern Expr *make_Expr(e_expr *);
extern b_expr *make_b_expr_of_Expr(Expr *);
extern b_expr *make_b_expr_of_Ite(Ite *);
extern b_expr *make_b_expr_of_Let(Let *);
extern Op *make_Op(op *,e_expr *,e_expr *);
extern Const *make_Const(int);
extern Id *make_Id(char *);
extern e_expr *make_e_expr_of_Id(Id *);
extern e_expr *make_e_expr_of_Const(Const *);
extern e_expr *make_e_expr_of_Op(Op *);
extern Plus *make_Plus();
extern op *make_op_of_Plus(Plus *);
```


Des définitions supplementaires :

```
/* Fait à la main */
#define makeLet(s,e,b) \
  (make_b_expr_of_Let(make_Let(s,e,b)))
#define makeIte(e,b1,b2) \
  (make_b_expr_of_Ite(make_Ite(e,b1,b2)))
#define makeExpr(e)    \
  (make_b_expr_of_Expr(make_Expr(e)))
#define makePlus(e1,e2)  \
  (make_e_expr_of_Op(make_Op(make_op_of_Plus(make_Plus
#define makeId(s) \
  (make_e_expr_of_Id(make_Id(s)))
#define makeConst(n)      \
  (make_e_expr_of_Const(make_Const(n)))
```


Le traducteur (en bison)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "lse_syntabstr.tic.h"

  extern void yyerror(char const *);
  extern int yylineno;
  extern FILE * yyin;
%}


%union {
  char* str;
  int num;
  union e_expr *e;
  union b_expr *b;
}
```

Le traducteur (en bison) : les types

```
%union {
  char* str;
  int num;
  union e_expr *e;
  union b_expr *b;
}

%left PLUS
%token LPAR RPAR
%token IF  ELSE
%token THEN
%token LET EQ IN
%token <str> ID
%token <num> CONST

%type <e> e_expression
%type <b> b_expression

%start S
```


Le traducteur (en bison) : la grammaire

```
%%
S: b_expression;

b_expression: e_expression {$$=makeExpr($1);}
| LPAR b_expression RPAR {$$=$2;}
| LET ID EQ e_expression IN b_expression
   {$$=makeLet($2,$4,$6);}
| IF e_expression THEN b_expression ELSE b_expression
   {$$=makeIte($2,$4,$6);}
;

e_expression: ID {$$=makeId($1);}
| CONST {$$=makeConst($1);}
| e_expression PLUS  e_expression
    {$$=makePlus($1,$3);}
| LPAR e_expression RPAR {$$=$2;}
;
%%
```


Le traducteur (en bison) : le main

```
void yyerror(char const *str)
{
  fprintf(stderr,"Ligne %d : %s\n", yylineno, str);
  exit(EXIT_FAILURE);
}
int
main(int argc, char* argv[])
{
  if ( argc > 1 ) {
    if((yyin = fopen( argv[1], "r" ))==NULL)
      exit(EXIT_FAILURE);
    yyparse();
    fclose(yyin);
  }
  else exit(EXIT_FAILURE);
  exit(EXIT_SUCCESS);
}
```


Plan

Grammaires L-attribuées

Grammaire *L*-attribuée.
L'évaluation des attributs peut se faire par un parcours en profondeur gauche (Left=gauche) de l'arbre dérivation.

- Tout attribut est synthétisé ou hérité
- Si

Production	Règle sémantique
⋮	
$A \to X_1 \dots X_{i-1} X_i \dots X_n$	⋮
	$X_i.b := f(c_1, \dots, c_k)$
	⋮
⋮	

alors $c_1, \dots, c_k$ sont des attributs de $X_1, \dots, X_{i-1}$,
ou des attributs hérités de A.

Attribution de types dans une déclaration

Production	Règle sémantique
$D \to T\ L$	$L.type := T.type$
$T \to$ `entier`	$T.type :=$ *entier*
$T \to$ `reel`	$T.type :=$ *reel*
$L \to L_1$ `,` `id`	$L_1.type := L.type$ *AjouterType*(`id`.*entree*, $L.type$)
$L \to$ `id`	*AjouterType*(`id`.*entree*, $L.type$)

Cette grammaire est *L*-attribuée.

Schémas de traduction

- comme une grammaire attribuée,
- les actions sont insérés
  à la droite de $\to$, n'importe où.
- Précise l'ordre des actions
  par rapport à un parcours en profondeur gauche de l'arbre.

Exemple : traduction postfixe des expressions additives

$E \to T\ R$
$R \to$ `addop` T {*print*(`addop`.*lexeme*)} $R \mid \varepsilon$
$T \to$ `chiffre` {*print*(`chiffre`.*val*)}

En bison . . .

. . . nous écrivons des schémas de traduction :

```
%union{
  int num;
  char *string;
%}
%token <string> addop <num> chiffre
%%
E: T R;
R: addop T {printf("%s",$1);} R | /* vide */;
T: chiffre {printf("%d",$1);} ;
%%
```

Contraintes

- si une action calcule un attribut de X_i alors
  se trouve à la gauche de X_i :

$$A \rightarrow \ldots \{\ X_i.b := f(\ldots)\ \} \ldots X_i \ldots$$

- si une action utilise un attribut de X_i alors
  elle se trouve à la droite de X_i :

$$A \rightarrow \ldots\ldots X_i \ldots \{\ b := f(\ldots X_i.c \ldots)\ \}$$

Élimination des reglès intermediaires

On peut toujours se ramener à une grammaire *L*-attribués :
Les schémas

$$E \rightarrow T\ R$$
$$R \rightarrow \texttt{addop}\ T\ \{\mathit{print}(\texttt{addop}.\mathit{lexeme})\}\ R \mid \varepsilon$$
$$T \rightarrow \texttt{chiffre}\ \{\mathit{print}(\texttt{chiffre}.\mathit{val})\}$$

$$E \rightarrow T\ R$$
$$R \rightarrow \texttt{addop}\ T\ M\ R \mid \varepsilon$$
$$M \rightarrow \varepsilon\ \{\mathit{print}(\texttt{addop}.\mathit{lexeme})\}$$
$$T \rightarrow \texttt{chiffre}\ \{\mathit{print}(\texttt{chiffre}.\mathit{val})\}$$

sont équivalents.

Schémas en Bison

Fichier `postfixe.y` :

```
%union{
  int num;
  char *string;
%}
%token <string> addop <num> chiffre
%%
E: T R;
R: addop T {printf("%s",$1);} R | /* vide */;
T: chiffre {printf("%d",$1);} ;
%%
```

Fichier `postfixe.output` :

```
0 $accept: E $end
1 E: T R
2 $@1: /* vide */
3 R: addop T $@1 R
4  | /* vide */
5 T: chiffre
```


Exercice

Considérez le fichier `interregles.y`

```
%union{
  int num;
  char *string;
}
%token <string> A <string> B
%%
start: A {puts($1);} B {puts($2);}
%%
```

Expliquez pourquoi nous obtenons :

```
$ bison interregles.y
interregles.y:7.30-31:
  $2 de start n'a pas de type déclaré
```

Traduction ascendante

- Nous savons comment calculer les attributs synthétisés.
- Pour calculer les attributs hérités :
  - on effectue les actions au moment des réductions,
  - Quand dans
    $$A \rightarrow X_1 \ldots, X_{i-1} X_i \ldots X_n$$
    on passe par X_i
    - $X_1, \ldots, X_{i-1}$ sont dans la pile,
    - A n'est pas dans la pile,
  - on cherchera un attribut herité de A dans la pile.

La declaration int p, q, r

Pile	Entrée	Action	
	int id, *id*, *id*\$		
int	*id*, *id*, *id*\$	$T \rightarrow int$	$type[ntop] := entier$
T	*id*, *id*, *id*\$		
T id	, *id*, *id*\$	$L \rightarrow id$	$ajouterType(entree[top], type[ntop])$
			$(type[ntop] = type[top-1])$
T L	, *id*, *id*\$		
T L,	*id*, *id*\$		
T L, *id*	, *id*\$	$L \rightarrow L, id$	$ajouterType(entree[top], type[ntop])$
			$type[top-2] := type[ntop]$
			$(type[ntop] = type[top-3])$
T L	, *id*\$		
T L,	*id*\$		
T L, *id*	\$	$L \rightarrow L, id$	$ajouterType(val[top], type[ntop])$
			$type[top-2] := type[ntop]$
			$(type[ntop] = type[top-3])$
T L	\$	$D \rightarrow T\ L$	$type[top] := type[top-1]$
D	\$		

Implantation de la grammaire attribuée pour les déclarations

Production	Règle sémantique
$D \rightarrow T\ L$	$val[top] := val[top-1]$
$T \rightarrow$ `entier`	$val[ntop] := entier$
$T \rightarrow$ `reel`	$val[ntop] := reel$
$L \rightarrow L_1$, `id`	$AjouterType(val[ntop], val[top-3])$ $/* \ val[top-2] := val[top-3] \ */$ (*inutile*)
$L \rightarrow$ `id`	$AjouterType(val[ntop], val[top-1])$

Chercher un attribut hérité dans la pile

Considérez

Production	Règle sémantique
$S \rightarrow a\ A\ C$	$C.he := A.sy$
$S \rightarrow b\ A\ B\ C$	$C.he := A.sy$
$C \rightarrow c$	$C.sy := f(C.he)$

Quand on réduit $C \rightarrow c$,
$C.he$ peut se trouver à distance -1 ou -2 du sommet la pile.

Pile	Entrée	Action	
$\ldots aAc$	$\ldots$	$C \to c$	$val.sy[ntop] := f(val.sy[top-1])$
			$(val.he[ntop] = val.sy[top-1])$
$\ldots aAC$	$\ldots$		
		$\vdots$	
$\ldots aAC$	$\ldots$	$S \to aAC$	$val.he[top] := val.sy[top-1]$
$\ldots S$			

Pile	Entrée	Action	
$\ldots bABc$	$\ldots$	$C \to c$	$val.sy[ntop] := f(val.sy[top-2])$
			$(val.he[ntop] = val.sy[top-2])$
$\ldots bABC$	$\ldots$		
		$\vdots$	
$\ldots bABC$	$\ldots$	$S \to bABC$	$val.he[top] := val.sy[top-2]$
$\ldots S$			

Autre exemple

Production	Règle sémantique
$B \to \alpha_0 A \alpha_1$	$A.he := B.he$
$C \to \beta_0 B \beta_1$	$B.he := C.he$
$E \to D\ C$	$C.he := D.i$

Pile	E.	A.
$\ldots D\beta_0\alpha_0 A\alpha_1$		$val.he[top-\lvert\alpha_1\rvert] := val[top-\lvert\beta_0\alpha_0 A\alpha_1\rvert]$
$\ldots D\beta_0 B$		
		$\vdots$
$\ldots D\beta_0 B\beta_1$		$val.he[top-\lvert\beta_1\rvert] := val[top-\lvert\beta_0 B\beta_1\rvert]$
$\ldots DC$		$val.he[top] := val[top-1]$
$\ldots E$		

Marqueurs et copies

On peut s'organiser pour que
la valeur d'un attribut herité se trouve en $val[ntop-1]$.

Production	Règle sémantique
$S \to a\,A\,C$	$C.he := A.sy$
$S \to b\,A\,B\,C$	$C.he := A.sy$
$C \to c$	$C.sy := f(C.he)$

Production	Règle sémantique
$S \to a\,A\,N\,C$	$C.he := N.sy$, $N.he := A.sy$
$N \to \varepsilon$	$N.sy := N.he$
$S \to b\,A\,B\,M\,C$	$C.he := M.sy$, $M.he := A.sy$
$M \to \varepsilon$	$M.sy := M.he$
$C \to c$	$C.sy := f(C.he)$

Marqueurs et copies

Production	Règle sémantique
$S \to a\,A\,N\,C$	$N.he := A.sy$, $C.he := N.sy$
$N \to \varepsilon$	$N.sy := N.he$
$S \to b\,A\,B\,M\,C$	$M.he := A.sy$, $C.he := M.sy$
$M \to \varepsilon$	$M.sy := M.he$
$C \to c$	$C.sy := f(C.he)$

devient

Production	Règle sémantique
$S \to a\,A\,N\,C$	$val.he[top-1] := val.sy[top-2]$,
	$val.he[top] := val.sy[top-1]$
$N \to \varepsilon$	$val.sy[top] := val.he[top-1]$
$S \to b\,A\,B\,M\,C$	$val.he[top-1] := val.sy[top-2]$,
	$val.he[top] := val.sy[top-1]$
$M \to \varepsilon$	$val.sy[ntop] := M.he[top-1]$
$C \to c$	$val.sy[top] := val.he[top-1]$

Remarques

- Le non-terminal N apparaît seulement dans les productions
$$S \rightarrow a\,A\,N\,C$$
$$N \rightarrow \varepsilon$$
Il connaît son contexte.
De même pour M.
- Cette transformations préserve les grammaires LL(1).
- Elle ne preserve pas les grammaires LR(1).
- Rappel :
$$LL(1) \subseteq LR(1)$$