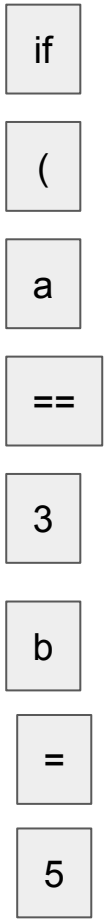
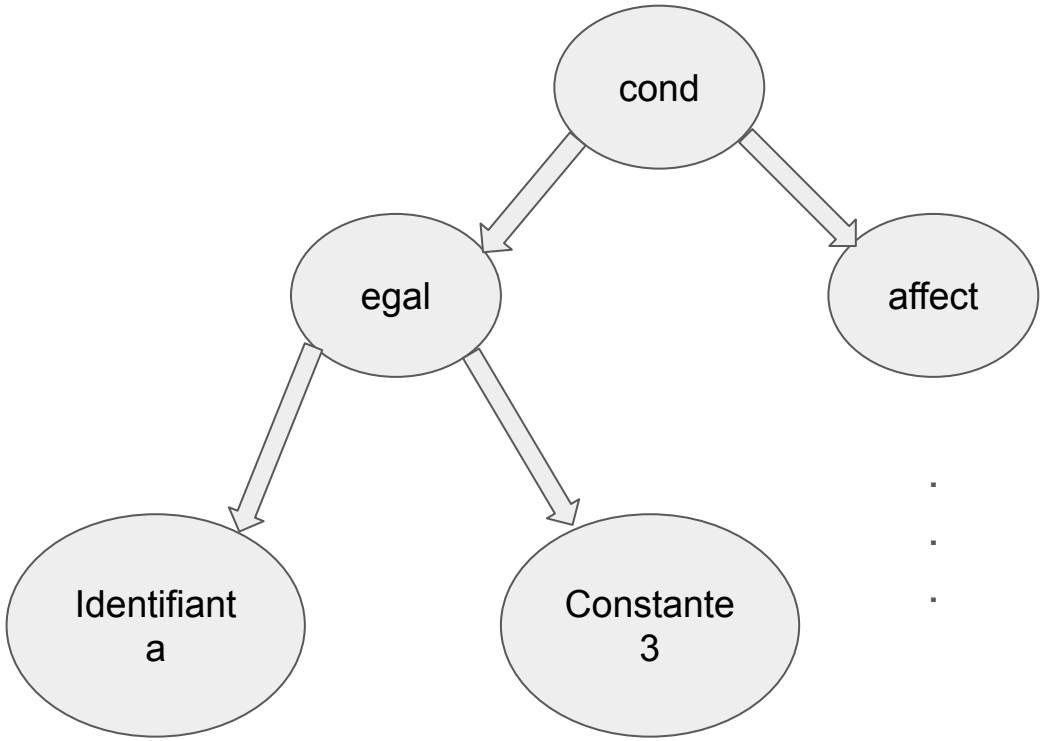
Notes cours Compilation

APP4 IIM

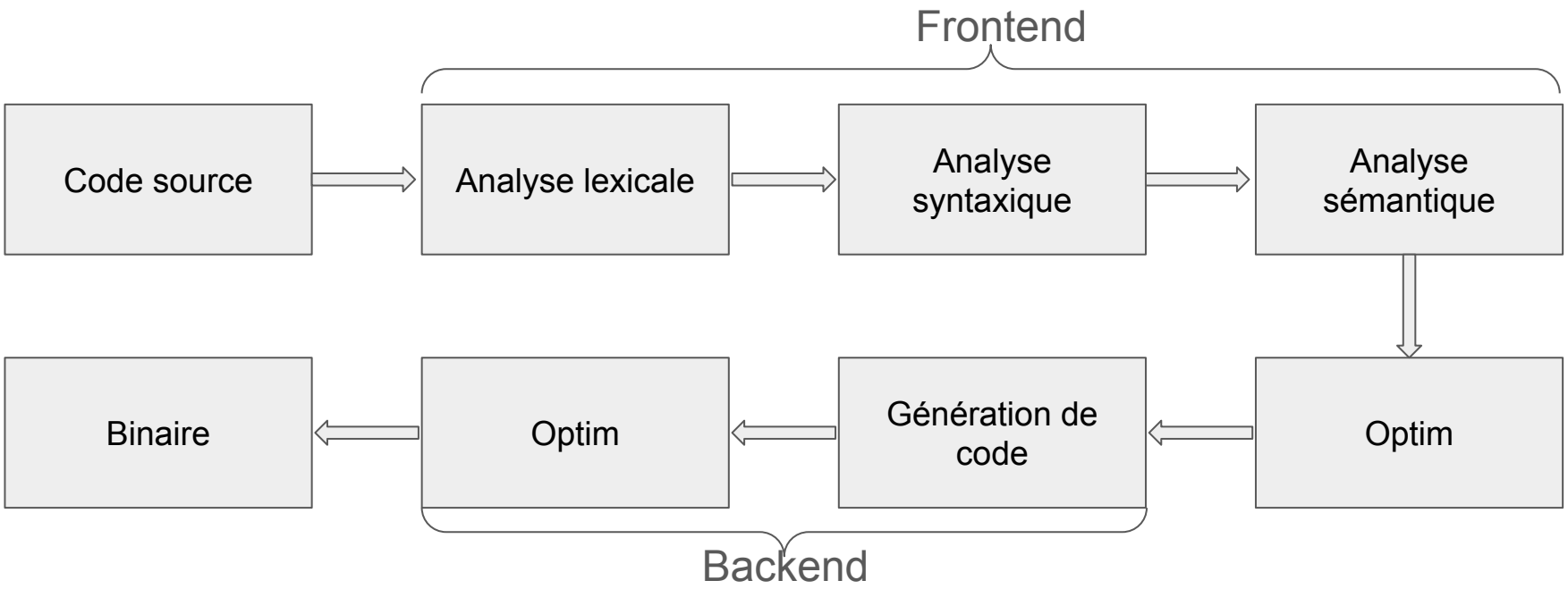
Diviser une chaîne de caractères en “tokens”

Ex : if(a==3)
b=5;





Fonctionnement du compilateur



Variables globales

token T, Last

Fonctions

```
init(...){initialisation; next();}
next(){...}
bool check(int type){if((T.type==type){next(); return true;}return false;}
accept(int type){if (!check(type)){erreur fatale("...");}}
```

Définir un token

```
enum{
        tok_eof,
        tok_if,
        tok_plus,
        .
        .
        .
}
```

```
Struct token{
        int type;
        int valeur;
        string chaine;
}
```

Contenu de la fonction next

```
next(){
        Last = T;
        [skip les espaces et commentaires]
        if (c'est un chiffre){lire un chiffre}
        else if(c'est une lettre){
                lire un identificateur
                Vérifier si c'est un mot clé
        }
        else{
                switch(c){...}
        }
}
```

Eof
const, ident
+, -, *, /, %, &&, ||, !
==, !=, <, >, <=, >=
(,), {, }, [,]
; = & ,
int void return if else do while
break continue
……..
Debug send recover

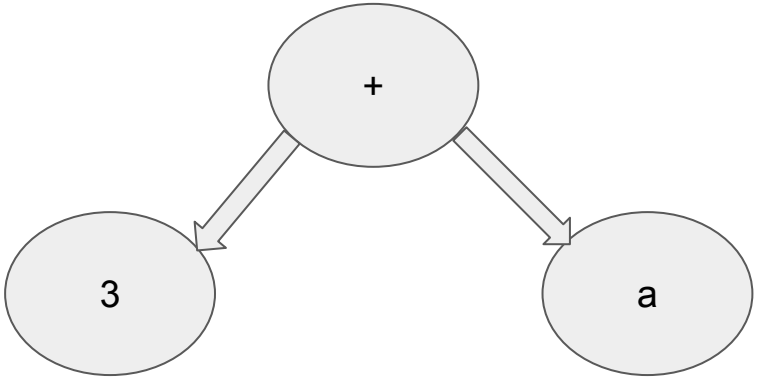
Les autres fonctions

```
main(){ //celle qui fera la compilation
        init(...)
        while(T.type!=tok_eof)
                gencode()
}

gencode(){
        A = optim;
}

optim(){
        A=AnaSem();
        return A;
}
```

```
AnaSem(){
        A=AnaSynt();
        return A;
}

AnaSynt(){
        A=....;
        return A.
}
```

Construction de l’arbre

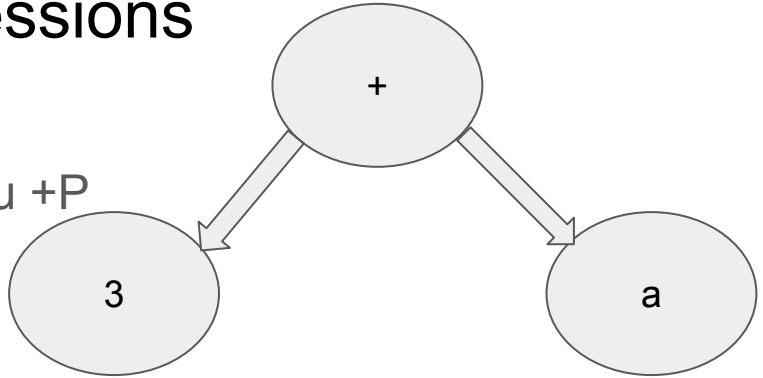
```
struct node {
        int type;
        int valeur;
        string chaine;
        int nbEnfants;
        node *enfants[];
}

node nd(int type)
ajouter_enfant(node parent, node enfant)
afficher(node A)
node node_v(int type, int valeur)
node node_1(int type, node a)
node node_2(int type, node a, node b)
```



Fonction E : analyse des expressions

```
node E(){
        return …P();
}

E←nb+E
Ou nb
Ou nb - E
.
.
.

Préfixe : ! - + * &
Suffixe : (...) [...]

Ex : -a[3] ← -(a[3])
```

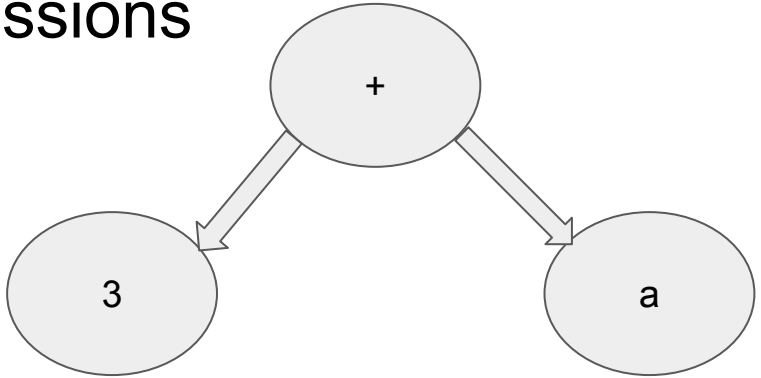
```
E←P
P←S ou !P ou -P ou +P
S←A
A←nb ou (E)
```




```
Node A(){
        if(check(tok_const){
                return node_v(nd_const, last.valeur)
                }
                else if(check(tok_par_open){
                        r=E();
                        accept(tok_par_close);
                        return r;
                }else{erreur_fatale()}
}
```

Fonction E : analyse des expressions

P→S—>S
P→!P—>not-–P
P→-P—>neg—P
P→+P—>P

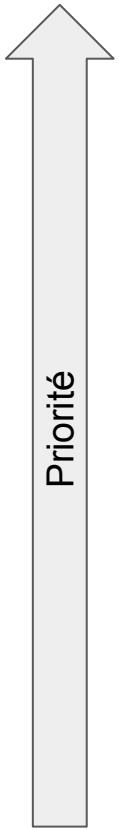



```
node P(){
	if(check(tok_not)){
		n=P();
		a=node_1(nd_not, n);
		return a;
	} else if(check(tok_neg)){
		return node_1(nd_neg, P());
	} else if(check(tok_plus)){return P();} else {
		return S();
	}
}
```

Fonction gencode

```
gencode(){
	A=optim();
	gennode(A)
}
gennode(node A){
	switch(A.type){
		case nd_const :
			print(“push”, A.valeur);
		case nd_not :
			gennode(A.enfants[0]);
			print(“not”);
		case nd_neg :
			print(“push 0”);
			gennode(A.enfants[0]);
			print(“sub”)
	}
}
```

Gestion des expressions



Opérateurs unaires (préfixes comme +, - et !)

6 : * / %

5 : + -

4 : == <= >= < > !=

3 : &&

2 : ||

1 : =

Gestion des expressions

```
E(int prio=){
        N=A();
        if est uneprio(T.type){
                op = T.type
                if tbl[op].prio<prio) break;
                next();
                M = E(tbl[op].pmin);
                R = node_2(op,N,M);
                return R;
        }
        return N;
}
```

tbl :

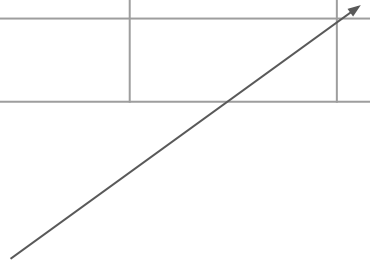
tok_plus	5	6
tok_nul	6	7
tok_and	3	4
…	…	…

Réécriture de E plus proprement

```
node *E(int prio){
	N=A();
	if (OP[T.type] != NULL){ // est un op binaire
		if (OP[T.type].prio < prio)
			return N;
		op = T.type; next();
		M = E(OP[op].parg); // au lieu de pmin
		R = node_2(OP[op].Ntype, N, M);
		return R;
	}
	return N;
}
```

OP :

tok_plus	5	6	nd_add
tok_nul	6	7	
tok_egal	1	1	nd_affect



associativité

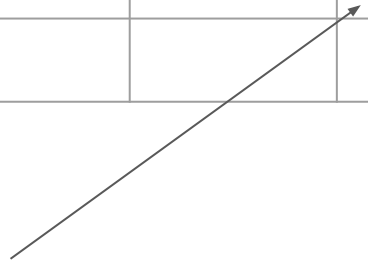
Quelques corrections… et voilà

```
node *E(int prio){
	N=P();
	while (OP[T.type] != NULL){
		if (OP[T.type].prio < prio)
			break;
		op = T.type; next();
		M = E(OP[op].parg); // au lieu de pmin
		N = node_2(OP[op].Ntype, N, M);
	}
	return N;
}
```

Donc : A()--> E(0)
Async appelle E(0)

OP :

tok_plus	5	6	nd_add
tok_mult	6	7	nd_mult
tok_egal	1	1	nd_affect



Modification de gennode (manière “dégeu”)

```
case nd_add :
        gennode(N→enfants[0])
        gennode(N→enfants[1])
        print (“add”); // pour sub c’est sub
case nd_mult :
        gennode(N→enfants[0])
        gennode(N→enfants[1])
        print(“mul”)
.
.
.
```

Modification de gennode (bonne manière)

```
gennode(node N){
	if (NF[N.type] != NULL){
		for (int i = 0; i < N.nbEnfants; i++){
			gennode(N.enfants[i])
		}
		print(NF[N.type].inst);
		return;
	}
	switch(N.type){
	}
}
```

NF :

nd_add	“add”	
nd_minus	“sub”	

Modification de gennode (bonne manière)

```
gennode(node N){
    if (NF[N.type] != NULL){
        print(NF[N.type].prefixe);
        for (int i = 0; i < N.nbEnfants; i++){
            gennode(N.enfants[i])
        }
        print(NF[N.type].suffixe);
        return;
    }
    switch(N.type){
    }
}
```

NF :

nd_add	“add”	
nd_minus	“sub”	“push 0”

Différence entre expressions et instructions

Expressions

3 + 4 * 2

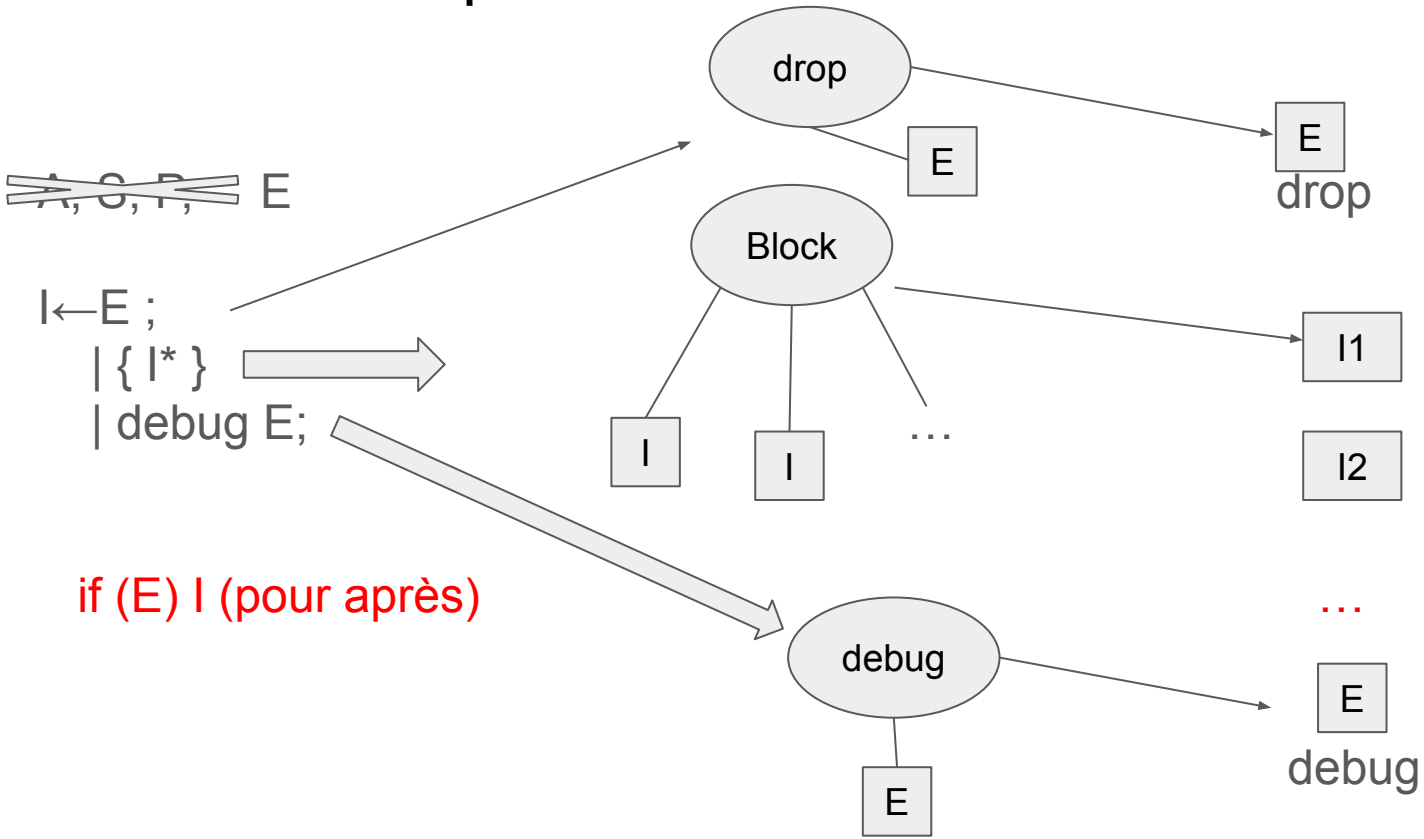
a = 5

Instructions

```
if(a==3)
        b = 5;
while (...){...}
```

a = 5; b = 3;

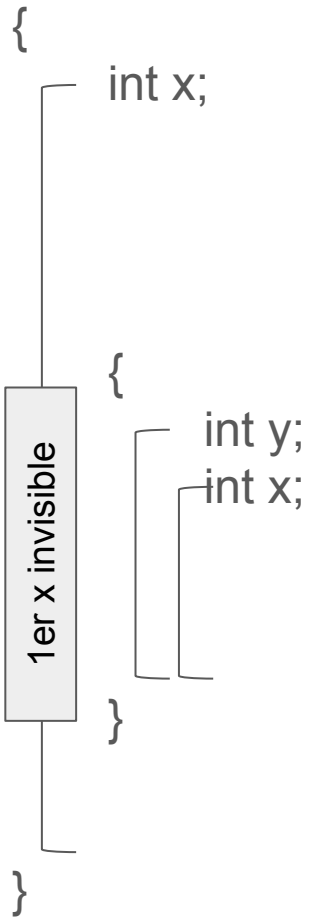
Différence entre expressions et instructions



Code à implémenter

```
node I(){
	if(check(tok_debug)) {
		N = E(0); accept(tok_ptvirgule); return node_1(nd_debug, N);
	} else if (check(tok_accoll_ouv){ N=node(nd_block);
		while(!check(tok_acc_ferm)){
			ajouterEnfant(N, I());
		}
		return N;
	} else {
		N = E(0); accept(tok_ptvirgule); return node_1(nd_drop, N);
	}
}
```

Visibilité des variables



Nouvelles fonctions

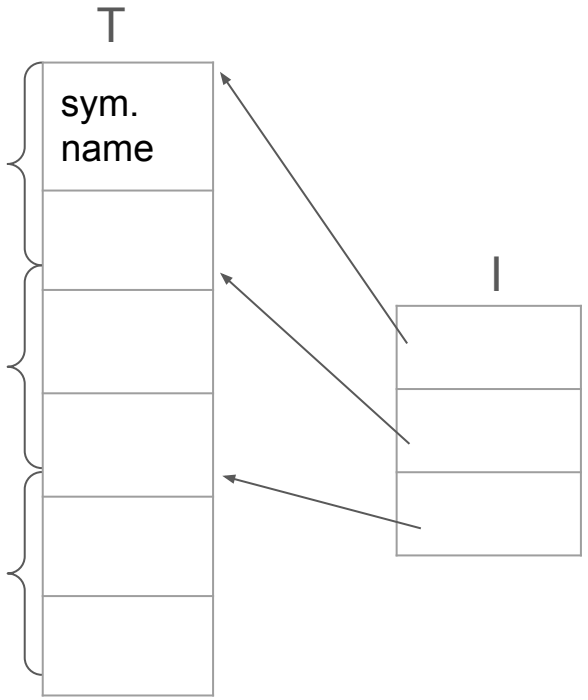
void begin() push(new table)
void end() pop()
sym declare(string name) //sym pour symbole
sym find(string name)

Nouvelles fonctions (structure pile)

```
declare(name){
	if(TS.top.find(name)) erreur_fatale(“ ”)
	sym=...
	TS.top[name] = sym;
	return sym;
}

find(name){
	for i = top to bottom{
		if(TS[i].find(name)){
			returnTS[i][name];
		}
	}
	erreur_fatale(“ ”)
}
```

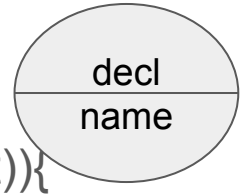
Nouvelles fonctions (tableaux T-symboles et I-indexes)

```
find(name){
	for(int i =top; i >=0; i--){
		if (T[i].name == name) return T[i]
	}
	erreur_fatale(“ ”);
}

begin(){I[top+1] = I[top]; top++;}
end(){top--;}

declare(name){
	for(i = debut du dernier block à fin du dernier block)
		if (T[i].name == name)
			erreur
	[ajout nom symbole]
}
```



Déclaration et utilisation de variables

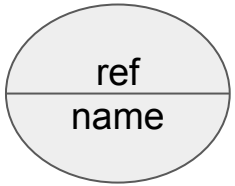
I ← … | int ident;



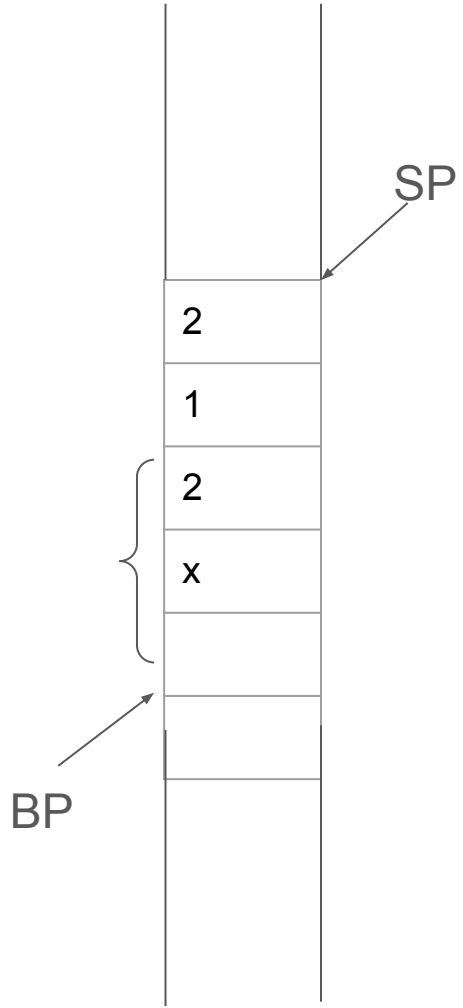
}else if (check(tok_int)){
    N = node_c(nd_decl, T.chaine);
    accept(tok_ident); accept(tok_ptvirgule);
    return N;}

1+x

1+2*x



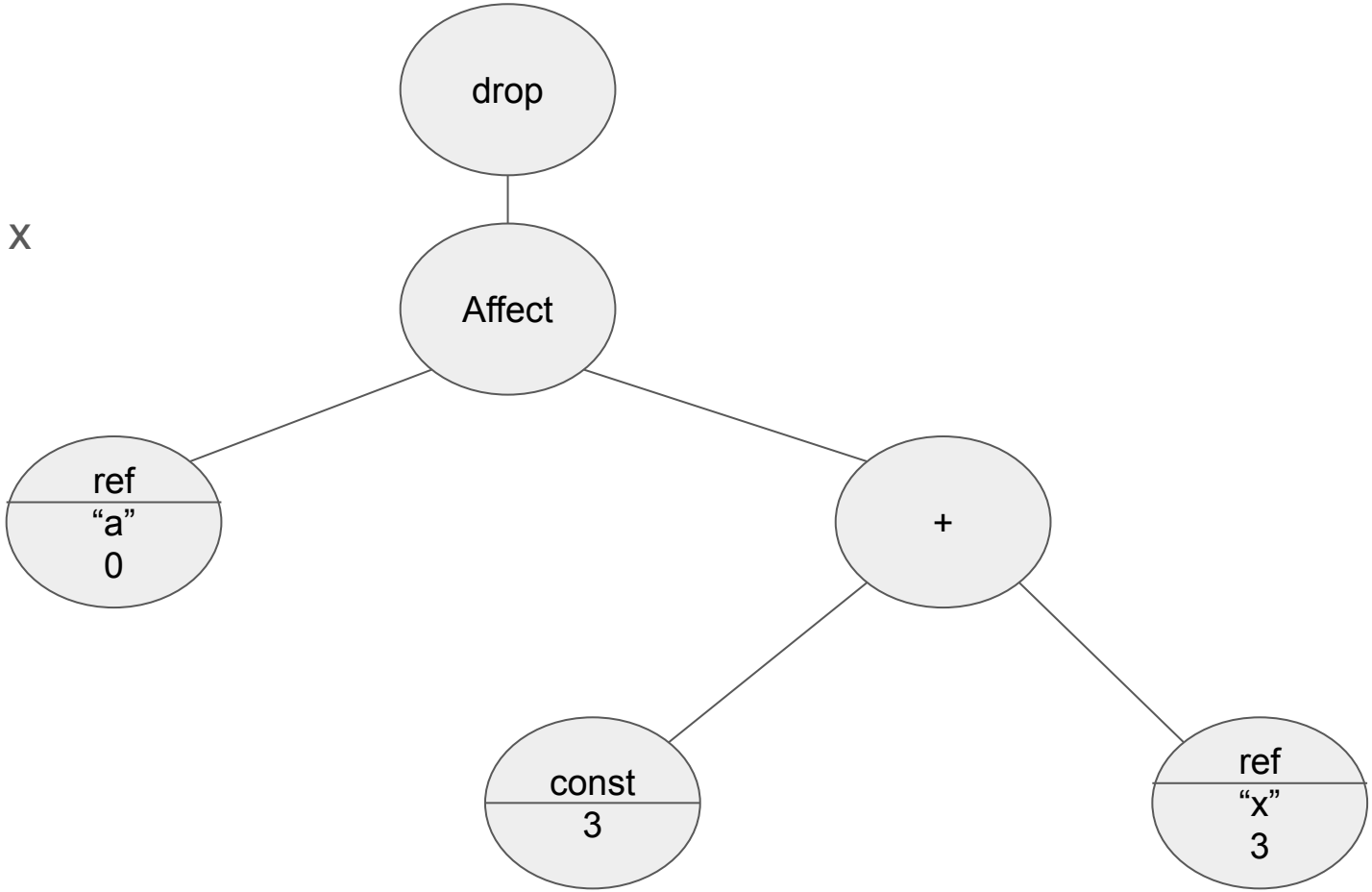
A← … | ident
    }else if(check(tok_ident) return node_c(nd_ref, Last.chaine);



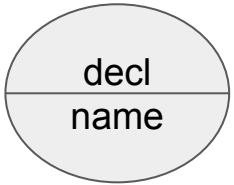
Exemple de construction d'un arbre

a = 3 + x

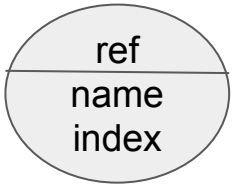
push 3
get 3
add
~~dup~~
set 0
~~drop~~



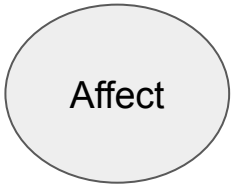
Génération du code correspondant aux noeuds

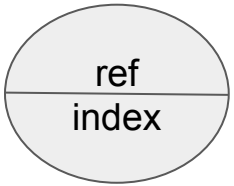


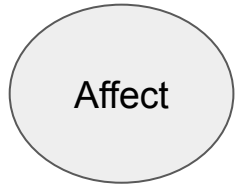
Pas de code

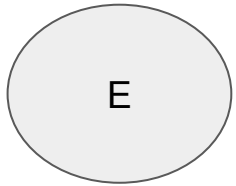


get N.index



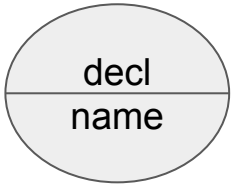




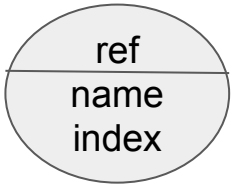


dup
set N.enfant[0].index

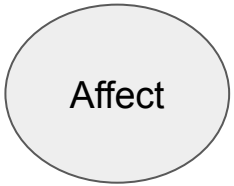
Génération du code correspondant aux noeuds

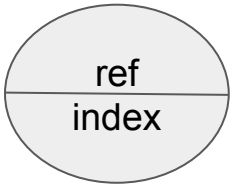


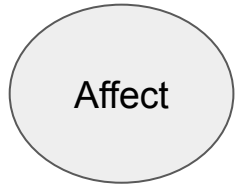
Pas de code

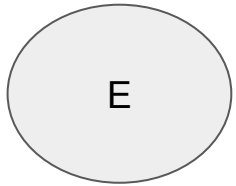


get N.index









dup
set N.enfant[0].index

Analyse sémantique

```
void SemNode(node N){
    switch(N.type){
        case nd_block :
            begin();
            for(i=0;i<N.nbEnfants;i++){
                SemNode(N.enfants[i]);
            }
            end();
        case nd_decl :
            S = declare(N.chaine);
            S.index = NbVar;
            NbVar++;
        case nd_ref : S = find(N.chaine); N.index = S.index;
        case nd_affect : if(N.enfant[0].type != nd_ref) erreur_fatale("");
            for(i=0;i<N.nbEnfants;i++){SemNode(N.enfants[i];}
        default : for(i=0;i<N.nbEnfants;i++){SemNode(N.enfants[i];}
    }}
```

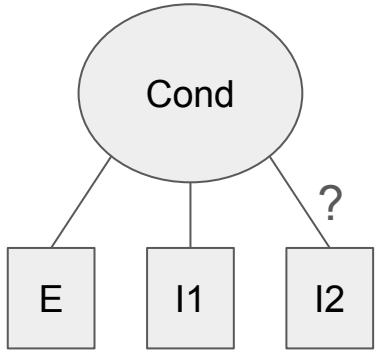
Analyse sémantique

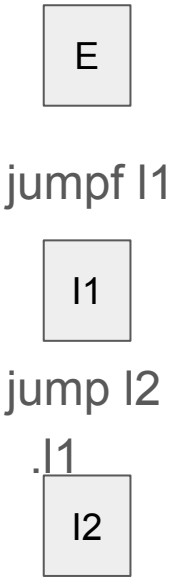
```
node AnaSem(){
        N = AnaSynt();
        NbVar = 0;
        SemNode(N);
        return N;
}

gencode(){
        N=optim();
        print(“resn”, NbVar);
        gennode(N);
        print(“drop”, NbVar);
}
```

Analyse syntaxique des boucles et des conditionnelles

I←…
    | if (E) I (else I)?
Code correspondant :

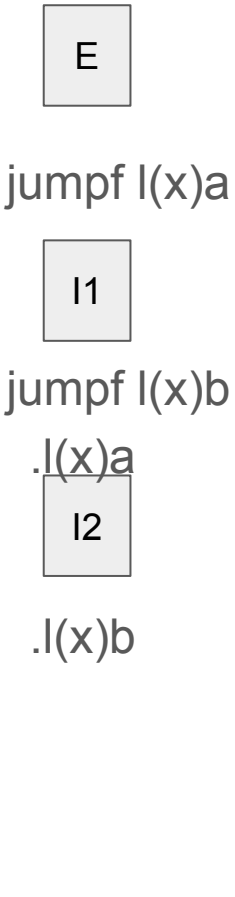
```
}else if (check(tok_if)){
        accept(tok_par_open);
        node E1 = E();
        accept(tok_par_close);
        node I1 = I();
        node I2 = NULL;
        if (check(tok_else)){I2 = I();}
        node N1 = node(nd_cond);
        ajouterEnfant(N1,E1);
        ajouterEnfant(N1,I1);
        ajouterEnfant(N1,I2);
        return N1;
}
```





Et s'il y a plusieurs if dans le code, on fait comment ?

```
nblbl = 0 //on compte le nombre de conditionnelles
case nd_cond :
        int l = nblbl++;
        gennode(N.enfants[0]);
        print("jumpf l",l,"a");
        gennode(N.enfants[1]);
        print("jump l",l,"b");
        print(".l",l,"a");
        gennode(N.enfants[2]);
        print(".l",l,"b");
```

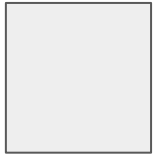


Code pour gérer les boucles

I←while(E) I | break; | continue;

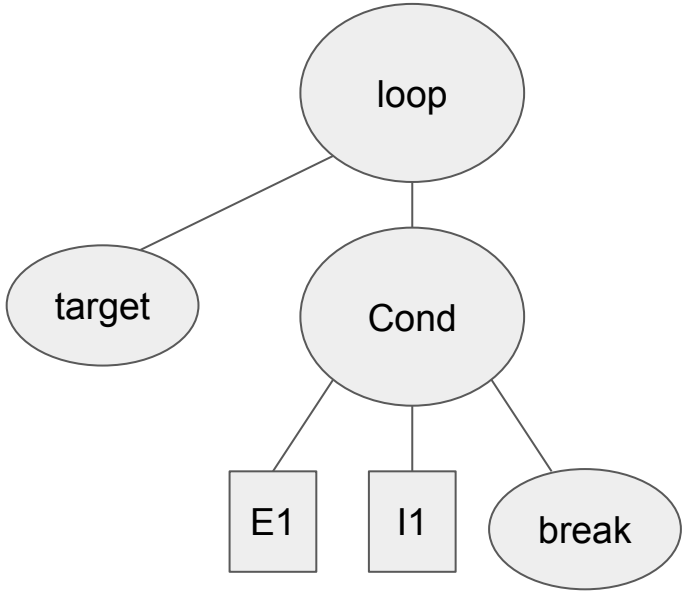
```
while(1){
        if(E1){
                I1
        } else{
                break;
        }
}
```

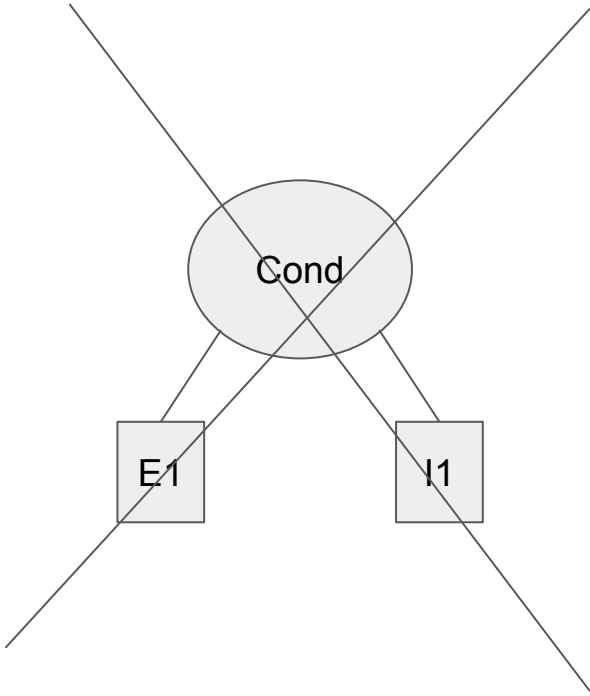
.I1



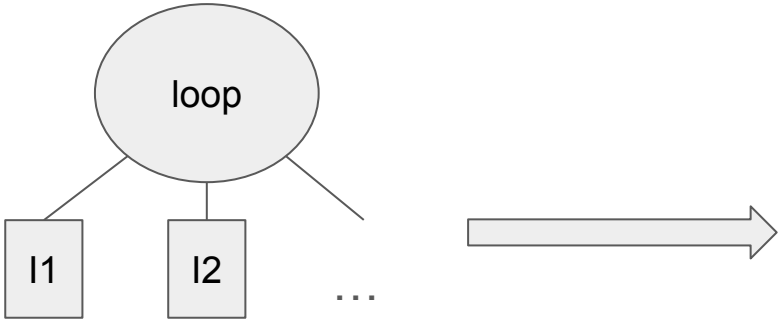
jump I1

.I2





Génération de code correspondante



.l(x)a
l0
l1
.
.
.
jump l(x)a
.l(x)b

target

break

continue

répéter
    Si E est vrai
        I
    Sinon
        c’est fini

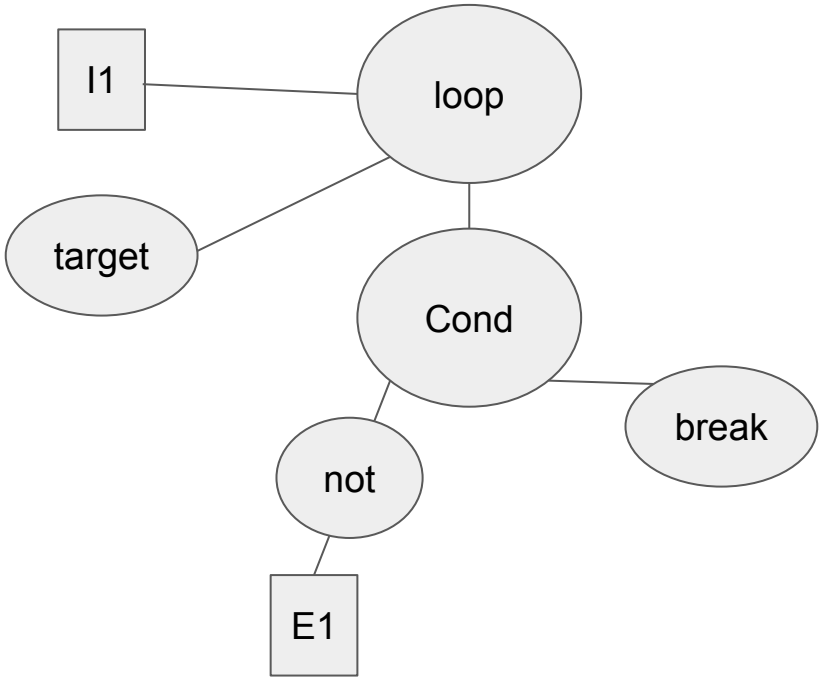
Génération de code correspondante

```
case nd_loop :
        int tmp = ll;
        ll = nblbl++;
        print(“.l”,ll,”a”);
        gennode(N.enfants[tous]);
        print(“jump l”,ll,”a”);
        print(“.l”,ll,”b”);
        ll = tmp;

case nd_break :
        print(“jump l”,ll,”b”);
```

Mais… problème avec des boucles imbriquées

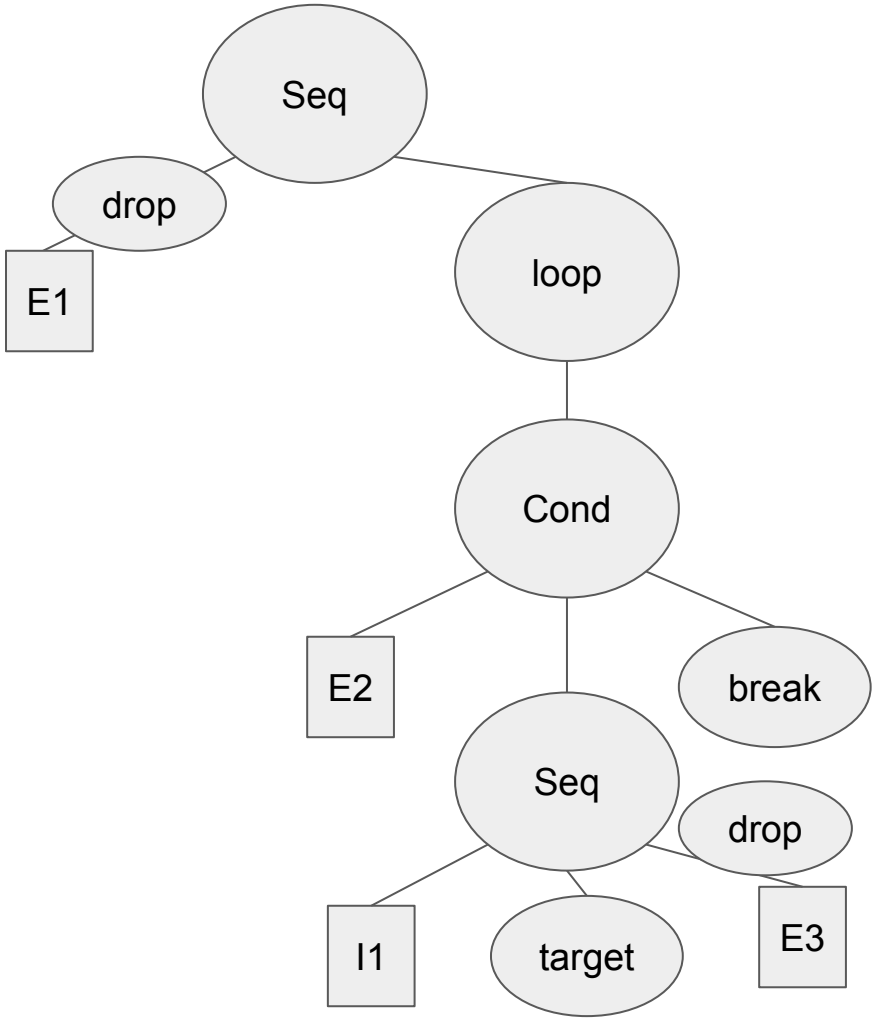
Code pour gérer les boucles

I←while(E) I | break; | continue;
| do I while (E);



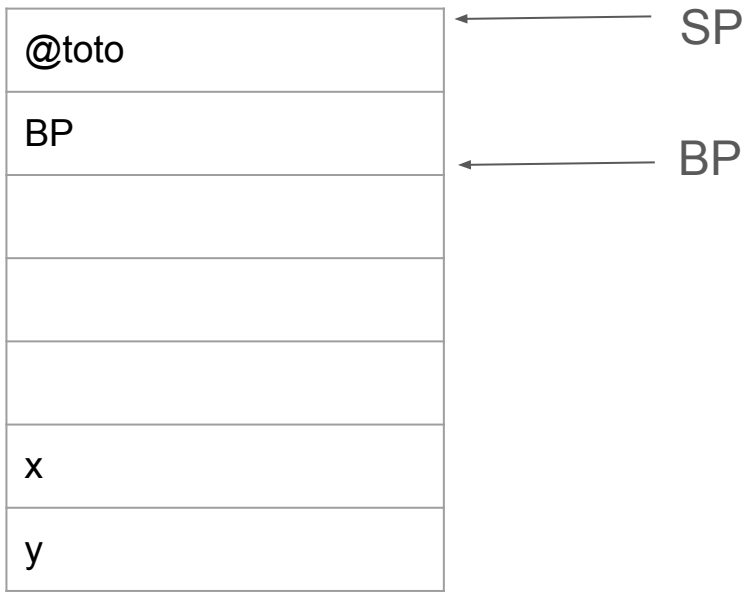
Code pour gérer les boucles

I←while(E) I | break; | continue;
| do I while (E);
| for(E1;E2;E3) I



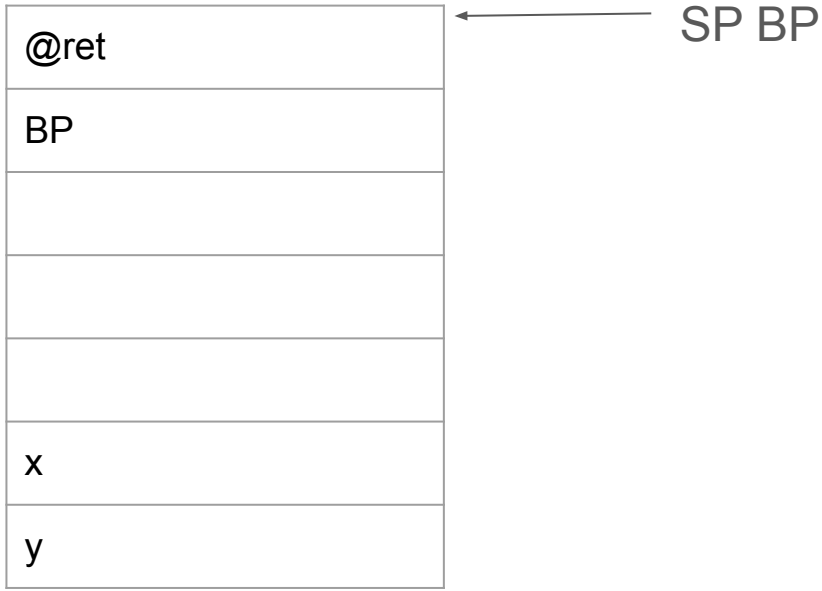
Gestion des fonctions

prep toto
call 0
ret



Gestion des fonctions

prep toto
call 0
ret



Gestion des fonctions

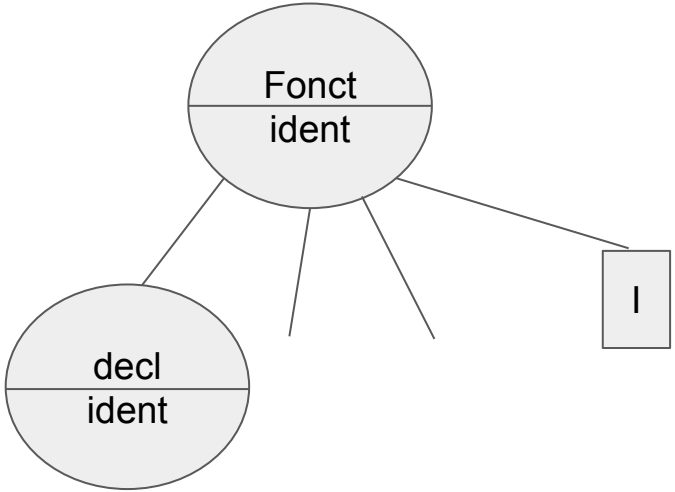
prep toto
call 0
ret

Flemme de réécrire le tableau, mais pop le dernier élément de la pile (résultat à retourner), puis SP = BP

Gestion des fonctions : analyseur syntaxique

F ← int ident ‘(‘ args ‘)’ I

(x(, x)*)?

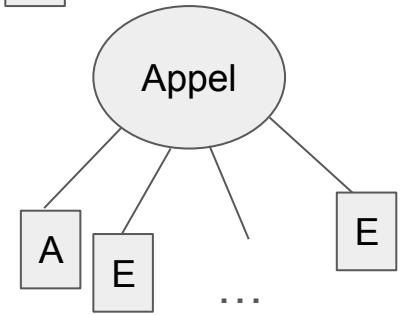



```
F(){
	accept(tok_int);
	accept(tok_ident); //Last_ident
	accept(tok_par_open);
	if(!check(tok_par_close)){
		do{
			accept(tok_int);
			accept(tok_ident);//Last_ident
		} while(check(tok_virgule));
	} accept(tok_par_close); I();//dernier enfant du noeud fonction
}
```

Gestion des fonctions : analyseur syntaxique

I ← … | return E;

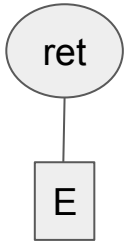
S← A (‘(‘ args ‘)’ ?)



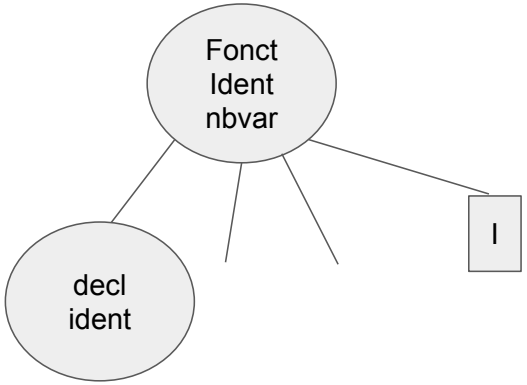
Gestion des fonctions : analyseur syntaxique

```
N = A();
if(check(tok_par_open)){
        N = node_1(nd_appel, N)
        //boucle pour check et récupérer tout
}
return N;
```

Gestion des fonctions : gencode



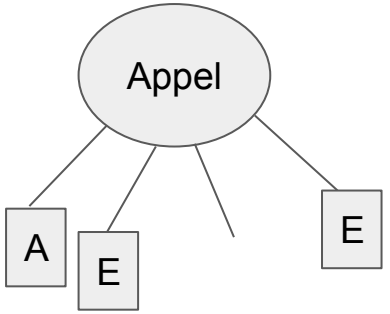
E
ret



.ident
res n nbvar

I

push 0
ret



prep enfants[0].ident

E1

E2

…
En
call nbenfants - 1

Gestion des fonctions : analyseur sémantique (bonus)

Semnode :

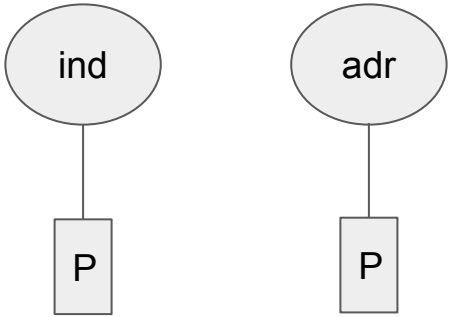
```
case nd_appel :
        //appels récursifs sur les enfants
        if(enfants[0].type_nd != var || find(enfants[0].ident).type != fonction) erreur_fatale(‘ ’);
        //cela suppose ajouter quelque chose qui permet de différencier une variable d’une
fonction, et potentiellement ajouter l’info sur le nb d’arguments de la fonction
case nd_fonction :
        declare(N.ident);
        Nbvar = 0;
        Debut bloc();
        //boucle sur les enfants
        Fin bloc();
        N.nbvar = nbvar - (N.nbEnfants-1);
```

Gestion des fonctions : à quoi ça ressemble dans le main ?

```
debutbloc();
while(T.type!=tok_eof){gencode();}
finbloc();
print(“.start”);
print(“prep main”);
print(“call 0”);
print(“halt”);
```

Ajout des pointeurs

P ← -P | !P | +P | *P | &P | S

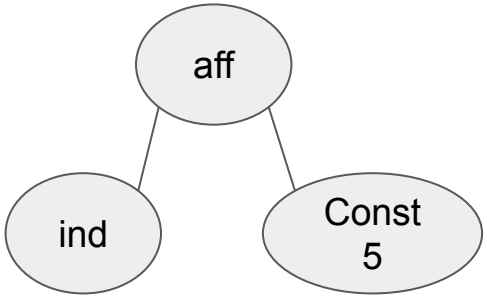


int *p;

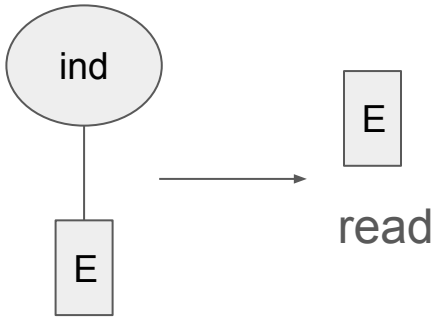
b = &p;

a = *p;

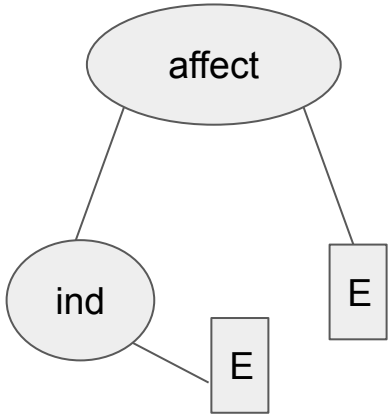
*p = 5;



Ajout des pointeurs : gencode

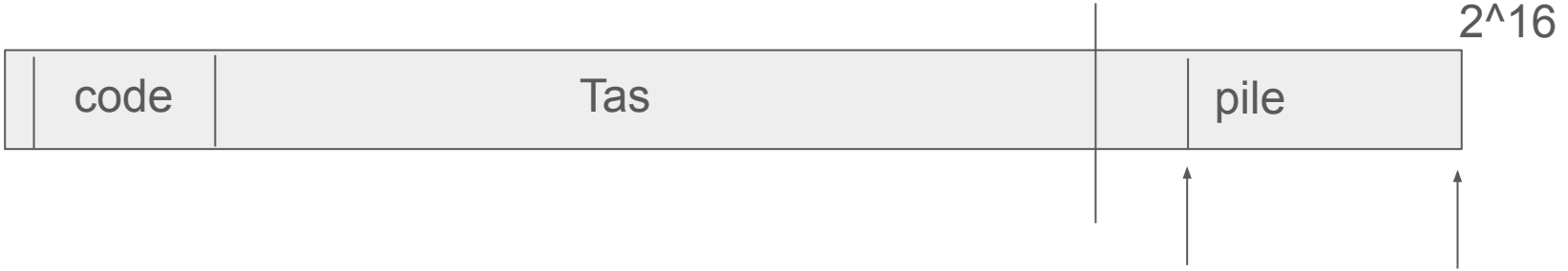


read




```
if (N.enfants[0].type == nd_ref){
…
} else if (N.enfants[0].type == nd_ind){
        gennode(N.enfants[1]);
        print(“dup”);
        gennode(N.enfants[0].enfants[0]);
        print(“write”);
}
```

Ajout des pointeurs : gestion de la mémoire - comment tester

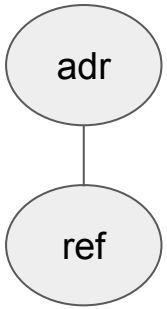



```
void free(int *p){}
int *malloc(int n){
        int *p;
        p = *0;
        *0 = *0 + n;
        return p;
}
```

```
int main(){
        int *p;
        p = malloc(5);
        *(p+2) = 10;
}
```

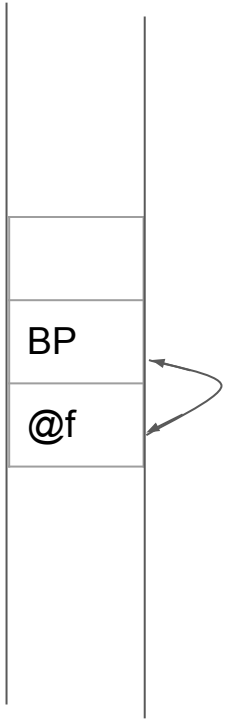
```
int main(){
        int *p;
        int x;
        p = &x;
}
```

Ajout des pointeurs : gencode pour les adresses




```
prep start
swap
drop 1
push 1
sub
N.enfants[0].index
sub
```

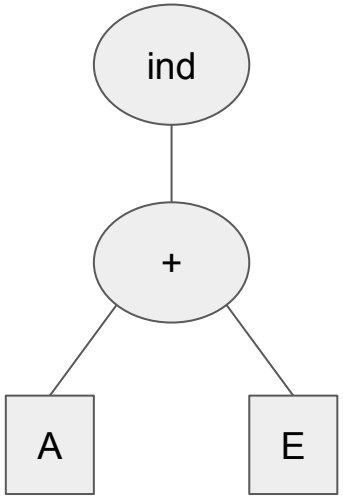
```
void swap(int *a, int *b){
        int t; t=*a;
        *a = *b; *b = t;
}
```



Ajout des pointeurs : gencode pour les adresses

S ← A ‘[‘ E ‘]’

```
} else if(check(tok_hook_open){
        …
}
```



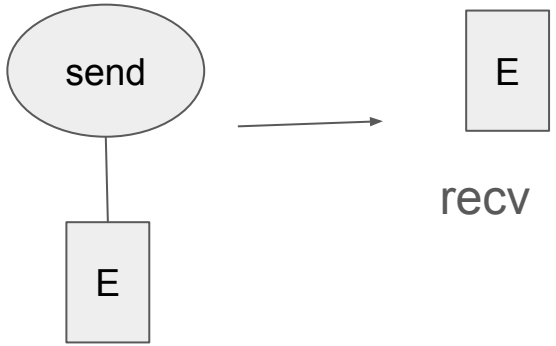
T[I]
=
*(T+I)
=
*(I+T)
=
I[T]

Communiquer avec le monde extérieur

A ← const | ident … | recv

Création d'un noeud recv, gencode : recv

I ← debug E; | send E;




```
void print(int n){
	int d; d = n%10;
	int r; r = n/10;
	if (r != 0) print(r);
	send d+48;
}
```

```
void println(int n) {print n; send 10;}
```

Informations sur le rendu

Date de rendu : 16 novembre 2025 à 23 h 59

Envoyer via l'adresse universitaire à : lavergne@lisn.fr

Contenu du rendu : Archive du code avec éventuellement les tests appliqués (.zip ou .tgz)

Penser à ajouter un README pour aider à se repérer (une sorte de carte)

Le rapport, pas trop volumineux avec l'essentiel (ce qui marche et ce qui ne marche pas)

Son format doit être .pdf