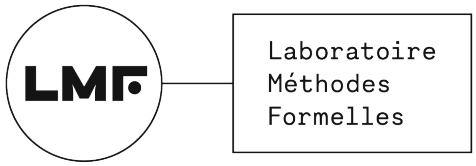


Introduction à la compilation

Polytech'Paris-Saclay– 4ème année –

Analyse Syntaxique Ascendante Deterministe

Burkhart Wolff (& Frederic Voisin)

Motivation

Motivation

- Sujet: méthodes d’analyse syntaxique déterministes (donc un fragment des grammaires type 2)

Motivation

- Sujet: méthodes d'analyse syntaxique déterministes (donc un fragment des grammaires type 2)
- Principe: construction des AST de manière ascendantes ("bottom-up")

Motivation

- Sujet: méthodes d'analyse syntaxique déterministes (donc un fragment des grammaires type 2)
- Principe: construction des AST de manière ascendantes ("bottom-up")
- Comme pour l'analyse lexique, il y a des générateurs de parsers a partir d'une grammaire

Motivation

- Sujet: méthodes d'analyse syntaxique déterministes (donc un fragment des grammaires type 2)
- Principe: construction des AST de manière ascendantes ("bottom-up")
- Comme pour l'analyse lexique, il y a des générateurs de parsers a partir d'une grammaire
- Strategie: remplacer le terminal "le plus a gauche" (LR strategy)

Exemple: Grammaire G_3

Exemple: Grammaire G_3

- Rappel G_3:

  $P(G_3) =$

Exemple: Grammaire G_3

- Rappel G_3:

  $P(G_3) =$

  E ::= E + T | T

Exemple: Grammaire G_3

- Rappel G_3:

```
P(G3) =
   E ::= E + T | T
 | T ::= T * F | F
```

Exemple: Grammaire G_3

- Rappel G_3:

$P(G_3) =$

```
   E ::= E + T | T
| T ::= T * F | F
| F ::= (E) | ident
```

Exemple: Grammaire G_3

- Rappel G_3:

```
P(G3)=
   E ::= E + T | T
 | T ::= T * F | F
 | F ::= (E) | ident
```

- Exemple: `ident+ident*ident`

Exemple: Grammaire G_3

- Rappel G_3:

  $P(G_3) =$

  E ::= E + T | T

  | T ::= T * F | F

  | F ::= (E) | ident

- Exemple: ident+ident*ident

  $\underline{E} \rightarrow E + \underline{T} \rightarrow E + \underline{T} * F \rightarrow E + \underline{T} *$ ident

  $\rightarrow E + \underline{F} *$ ident $\rightarrow \underline{E} +$ ident $*$ ident

  $\rightarrow \underline{T} +$ ident $*$ ident

  $\rightarrow \underline{F} +$ ident $*$ ident

  $\rightarrow$ ident + ident * ident

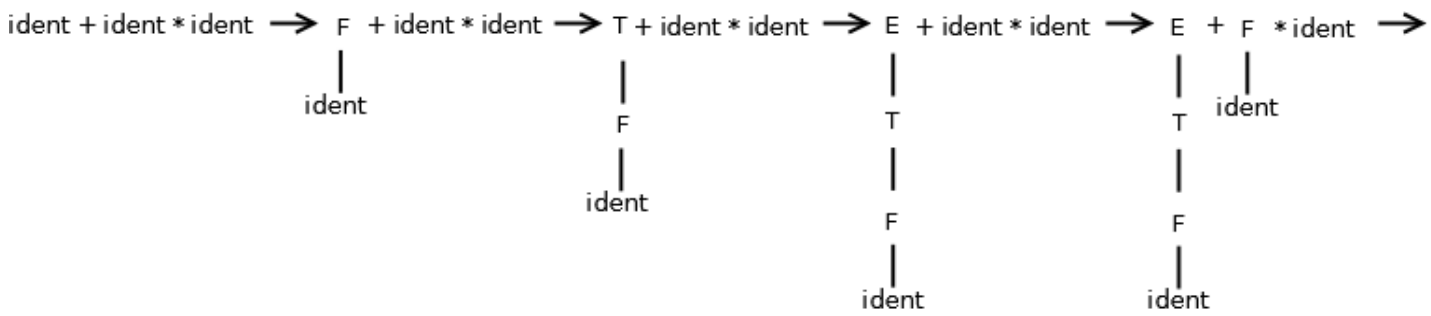
Exemple: Grammaire G_3

Exemple: Grammaire G_3

- Derivations G_3:

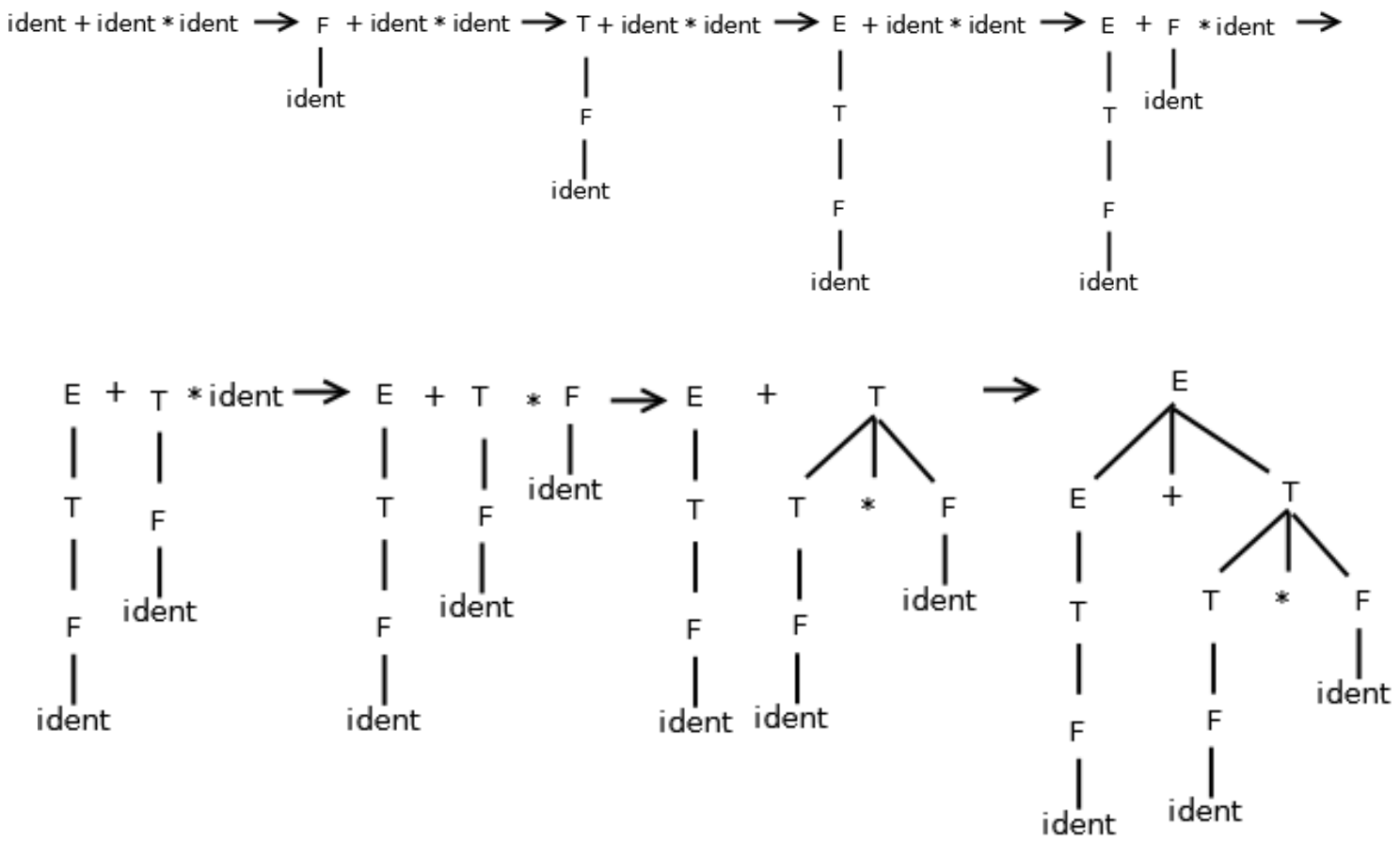
Exemple: Grammaire G_3

- Derivations G_3:



Exemple: Grammaire G_3

- Derivations G_3:



Définitions de base

- Objectif: trouver méthode d'analyse syntaxique déterministe (donc un fragment des grammaires type 2)

- Concept important (prerequisite):

Définitions :

$$\textit{First}(X) = \{\, a \in V_t \mid \exists\, \alpha \in V_t^* : X \rightarrow^* a\,\alpha \,\} \cup \{\, \varepsilon \,\} \quad \text{si } X \rightarrow^* \varepsilon,$$
$$\{\, a \in V_t \mid \exists\, \alpha \in V_t^* : X \rightarrow^* a\,\alpha \,\} \quad \text{sinon.}$$

$$\textit{Follow}(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

First-Sets : Examples

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $
E ::= T E'
E'::= + T E' | ε
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                    First(S)  = { ident, ( }
E ::= T E'
E'::= + T E' | ε
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                    First(S)  = { ident, ( }
E ::= T E'                   First(E)  = { ident, ( }
E'::= + T E' | ε
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                 First(S)  = { ident, ( }
E ::= T E'                First(E)  = { ident, ( }
E'::= + T E' | ε          First(E') = { +, ε }
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                 First(S)  = { ident, ( }
E ::= T E'                First(E)  = { ident, ( }
E'::= + T E' | ε          First(E') = { +, ε }
T ::= F T'                First(T)  = { ident, ( }
T'::= * F T' | ε
F ::= ident | ( E )
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                 First(S)  = { ident, ( }
E ::= T E'                First(E)  = { ident, ( }
E'::= + T E' | ε          First(E') = { +, ε }
T ::= F T'                First(T)  = { ident, ( }
T'::= * F T' | ε          First(T') = { *, ε }
F ::= ident | ( E )
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                 First(S)  = { ident, ( }
E ::= T E'                First(E)  = { ident, ( }
E'::= + T E' | ε          First(E') = { +, ε }
T ::= F T'                First(T)  = { ident, ( }
T'::= * F T' | ε          First(T') = { *, ε }
F ::= ident | ( E )       First(F)  = { ident, ( }
```

First-Sets : Examples

- First - Sets of G_2

G_2:

```
S ::= E $                 First(S)  = { ident, ( }
E ::= T E'                First(E)  = { ident, ( }
E'::= + T E' | ε          First(E') = { +, ε }
T ::= F T'                First(T)  = { ident, ( }
T'::= * F T' | ε          First(T') = { *, ε }
F ::= ident | ( E )       First(F)  = { ident, ( }
```

- First - Sets of G_3

G_3:

```
S ::= E $                 First(S) = { ident, ( }
E ::= E + T | T           First(E) = { ident, ( }
T ::= T * F | F           First(T) = { ident, ( }
F ::= ident | ( E )       First(F) = { ident, ( }
```

Un Algorithme

- Nous commençons par calculer *First*, à valeurs dans $V_t \cup \{ \varepsilon \}$, en étendant sa définition à toute séquence de symboles avec les définitions par induction suivantes ($a \in V_t$, $X \in V_N$, $\alpha \in (V_t \cup V_N)^*$)

$First(\varepsilon) = \{ \varepsilon \}$

$First(a) = \{ a \}$

$First(X) = \bigcup First(\alpha)$, union sur toutes les règles de la forme $X ::= \alpha$ dans G

$First(a\alpha) = \{ a \}$

$First(X\alpha) = \begin{cases} (First(X) - \{ \varepsilon \}) \cup First(\alpha) & \text{si } X \rightarrow^* \varepsilon \\ First(X) & \text{sinon.} \end{cases}$

Follow-Sets : Examples

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{ a \in V_t \mid \exists \alpha, \beta \in V_t^* : S \rightarrow^* \alpha X a \beta \}$$

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{ a \in V_t \mid \exists \alpha, \beta \in V_t^* : S \rightarrow^* \alpha X a \beta \}$$

- Follow - Sets of G_2

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $
E ::= T E'
E'::= + T E' | ε
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $                    Follow(S) = ∅
E ::= T E'
E'::= + T E' | ε
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $                Follow(S) = ∅
E ::= T E'               Follow(E) = {$, )}
E'::= + T E' | ε
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $                 Follow(S) = ∅
E ::= T E'                Follow(E) = {$, )}
E'::= + T E' | ε          Follow(E')= {$, )}
T ::= F T'
T'::= * F T' | ε
F ::= ident | ( E )
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $                Follow(S) = ∅
E ::= T E'               Follow(E) = {$, )}
E'::= + T E' | ε         Follow(E')= {$, )}
T ::= F T'               Follow(T) = { +, $, )}
T'::= * F T' | ε
F ::= ident | ( E )
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $                Follow(S) = ∅
E ::= T E'               Follow(E) = {$, )}
E'::= + T E' | ε         Follow(E')= {$, )}
T ::= F T'               Follow(T) = { +, $, )}
T'::= * F T' | ε         Follow(T')= {$, ), +}
F ::= ident | ( E )
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $              Follow(S) = ∅
E ::= T E'             Follow(E) = {$, )}
E'::= + T E' | ε       Follow(E')= {$, )}
T ::= F T'             Follow(T) = { +, $, )}
T'::= * F T' | ε       Follow(T')= {$, ), +}
F ::= ident | ( E )    Follow(F) = {*, ), $, +}
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $              Follow(S) = ∅
E ::= T E'             Follow(E) = {$, )}
E'::= + T E' | ε       Follow(E')= {$, )}
T ::= F T'             Follow(T) = { +, $, )}
T'::= * F T' | ε       Follow(T')= {$, ), +}
F ::= ident | ( E )    Follow(F) = {*, ), $, +}
```

- Follow - Sets of G_3

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $              Follow(S) = ∅
E ::= T E'             Follow(E) = {$, )}
E'::= + T E' | ε       Follow(E')= {$, )}
T ::= F T'             Follow(T) = { +, $, )}
T'::= * F T' | ε       Follow(T')= {$, ), +}
F ::= ident | ( E )    Follow(F) = {*, ), $, +}
```

- Follow - Sets of G_3

G3:

```
S ::= E $
E ::= E + T | T
T ::= T * F | F
F ::= ident | '(' E ')'
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $              Follow(S) = ∅
E ::= T E'             Follow(E) = {$, )}
E'::= + T E' | ε       Follow(E')= {$, )}
T ::= F T'             Follow(T) = { +, $, )}
T'::= * F T' | ε       Follow(T')= {$, ), +}
F ::= ident | ( E )    Follow(F) = {*, ), $, +}
```

- Follow - Sets of G_3

G3:

```
S ::= E $                    Follow(S) = ∅
E ::= E + T | T
T ::= T * F | F
F ::= ident | '(' E ')'
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $              Follow(S) = ∅
E ::= T E'             Follow(E) = {$, )}
E'::= + T E' | ε       Follow(E')= {$, )}
T ::= F T'             Follow(T) = { +, $, )}
T'::= * F T' | ε       Follow(T')= {$, ), +}
F ::= ident | ( E )    Follow(F) = {*, ), $, +}
```

- Follow - Sets of G_3

G3:

```
S ::= E $                  Follow(S) = ∅
E ::= E + T | T            Follow(E) ={$,+,)}
T ::= T * F | F
F ::= ident | '(' E ')'
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $                Follow(S) = ∅
E ::= T E'               Follow(E) = {$, )}
E'::= + T E' | ε         Follow(E')= {$, )}
T ::= F T'               Follow(T) = { +, $, )}
T'::= * F T' | ε         Follow(T')= {$, ), +}
F ::= ident | ( E )      Follow(F) = {*, ), $, +}
```

- Follow - Sets of G_3

G3:

```
S ::= E $                    Follow(S) = ∅
E ::= E + T | T              Follow(E) ={$,+,)}
T ::= T * F | F              Follow(T) ={$,+,*,)}
F ::= ident | '(' E ')'
```

Follow-Sets : Examples

- Rappel Definition:

$$Follow(X) = \{\, a \in V_t \mid \exists\, \alpha, \beta \in V_t^* : S \rightarrow^* \alpha\, X\, a\, \beta \,\}$$

- Follow - Sets of G_2

G_2:

```
S ::= E $              Follow(S) = ∅
E ::= T E'             Follow(E) = {$, )}
E'::= + T E' | ε       Follow(E')= {$, )}
T ::= F T'             Follow(T) = { +, $, )}
T'::= * F T' | ε       Follow(T')= {$, ), +}
F ::= ident | ( E )    Follow(F) = {*, ), $, +}
```

- Follow - Sets of G_3

G3:

```
S ::= E $                  Follow(S) = ∅
E ::= E + T | T            Follow(E) ={$,+,)}
T ::= T * F | F            Follow(T) ={$,+,*,)}
F ::= ident | '(' E ')'    Follow(F) ={$,+,*,)}
```

Calculer Follow-Sets

On calcule *Follow* par application itérée des relations suivantes aux règles de G :

si $X ::= \alpha\ Y\ \beta \in G$, *pour tout* a dans *First*(β), $a \neq \varepsilon$, *alors* $a \in$ *Follow*(Y)

si $X ::= \alpha\ Y\ \beta \in G$ *et* $\beta \rightarrow^* \varepsilon$, *pour tout* a *dans Follow*(X), alors $a \in$ *Follow*(Y)

si $X ::= \alpha\ Y \in G$, *pour tout* a *dans Follow*(X), *alors* $a \in$ *Follow*(Y)

Shift

Shift

- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

Shift

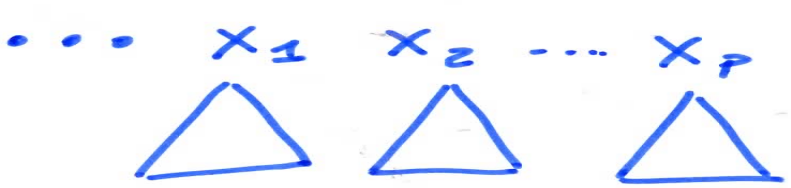
- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$

Shift

- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

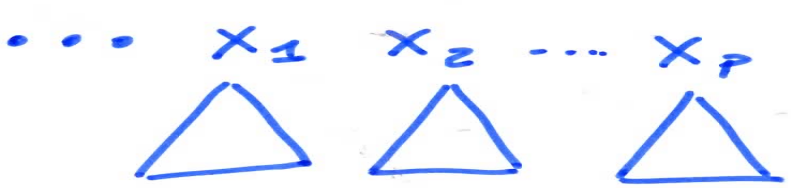
Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$



Shift

- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$



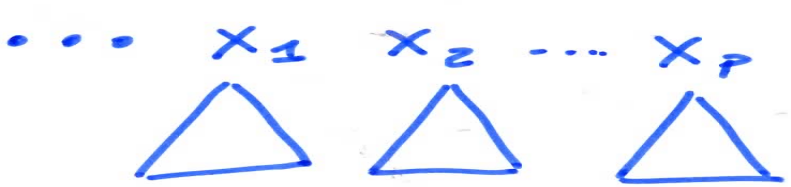
Après empilement :

$\mathbf{w_i}$ | Mot restant: $\mathbf{w_{i+1} \ldots w_n}$

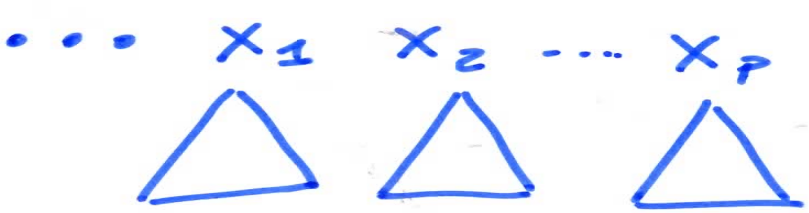
Shift

- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$



Après empilement :

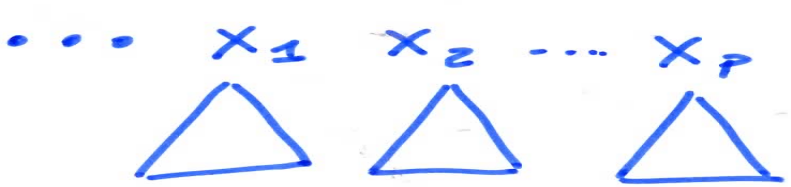


w_i | Mot restant: $\mathbf{w_{i+1} \ldots w_n}$

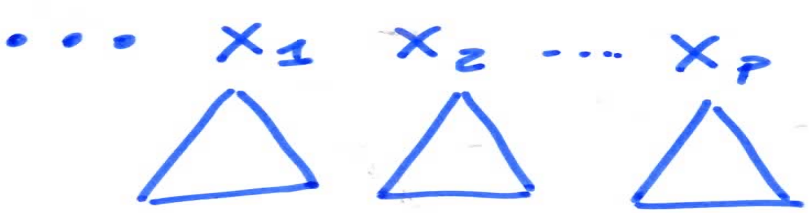
Shift

- le premier mouvement possible consiste à **empiler** (*shift*) la lettre courante du mot pour en faire un arbre réduit à sa racine, et on avance dans le mot. Ci-dessous, dans le dessin résumant la situation après empilement, w_i doit être vu comme un arbre réduit à sa racine

Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$



Après empilement :



w_i | Mot restant: $\mathbf{w_{i+1} \ldots w_n}$

Reduce

Reduce

- dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu. Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

Reduce

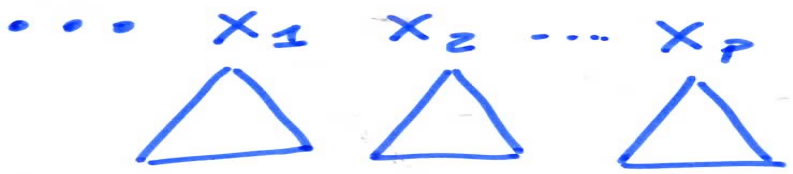
- dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu. Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$

Reduce

- dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu. Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

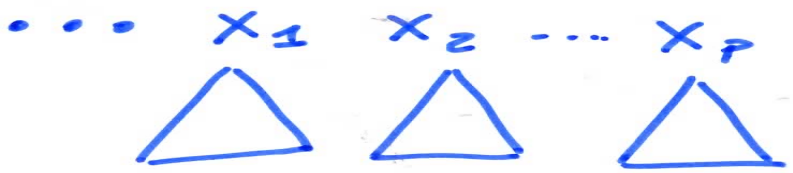
Au départ : Pile d'analyse | Mot restant: $w_i w_{i+1} \ldots w_n$



Reduce

- dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu. Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

Au départ : Pile d'analyse | Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$



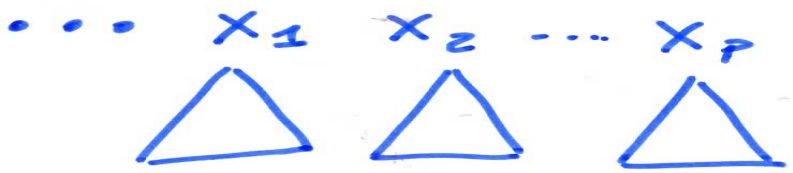
Après réduction par la production $X ::= X_1 \ldots X_p$:

| Mot restant: $\mathbf{w_i w_{i+1} \ldots w_n}$

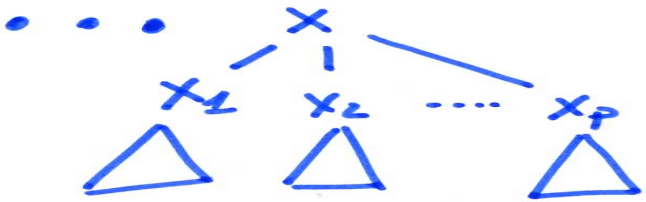
Reduce

- dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu. Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

Au départ : Pile d'analyse | Mot restant: $w_i w_{i+1} \ldots w_n$



Après réduction par la production $X ::= X_1 \ldots X_p$:

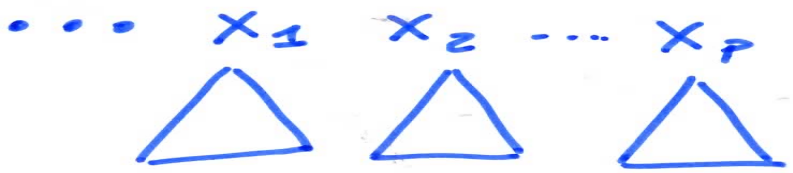


| Mot restant: $w_i w_{i+1} \ldots w_n$

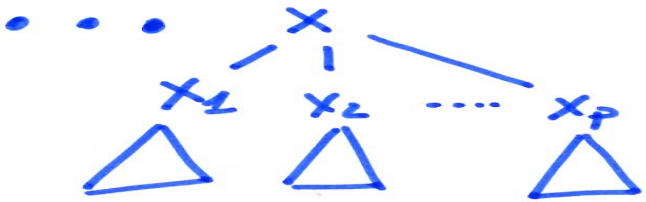
Reduce

- dépiler des arbres (dont la séquence des racines forme une **partie droite de production**), à les **enraciner** (les « **réduire** », *reduce* en anglais) sous une nouvelle racine dont l'étiquette est la partie gauche de cette production et à empiler l'arbre ainsi obtenu. Le mot restant à analyser est sans changement. **Seules les étiquettes des racines importent dans la pile pas les sous-arbres**

Au départ : Pile d'analyse | Mot restant: $w_i w_{i+1} \ldots w_n$



Après réduction par la production $X ::= X_1 \ldots X_p$:



| Mot restant: $w_i w_{i+1} \ldots w_n$

Shift / Reduce Conflicts

Shift / Reduce Conflicts

- **choisir entre empiler et réduire** quand les deux possibilités existent, c'est-à-dire quand une partie droite de production apparaît en sommet de pile ;

Shift / Reduce Conflicts

- **choisir entre empiler et réduire** quand les deux possibilités existent, c'est-à-dire quand une partie droite de production apparaît en sommet de pile ;
- Il faut notamment savoir choisir quand faire des réductions par des productions dont la partie droite est ε puisque le mot vide est toujours virtuellement présent en sommet de pile !

Reduce/Reduce Conflicts

- **choisir entre plusieurs réductions possibles** lorsque en sommet de pile on a une partie droite de production à laquelle on peut associer deux parties gauches distinctes
- Par exemple: `N ::= ε` et `M ::= ε` ou quand une partie droite de production est **suffixe** d'une autre partie droite
- Cas de `E ::= E+T` et `E := T`, avec `E+T` présent en sommet de pile : réduit-on `E+T` en `E` ou seulement `T` en `E` ?

Towards LR(0) Automata

- Items

Définition : on appelle **item LR(0)**, un élément de la forme [X: α . β] où X::= αβ est une règle de grammaire de G (à condition que αβ≠ε).

Si αβ=ε, l'item correspondant est [X:.]

- On mets les items C::=α.Xβ et C'::=α'.Yβ' dans un classe d'équivalence si $X \rightarrow^* Y\beta$''.
  On peut grouper les classes d'équivalence d'items dans un automate dite LR(0) comme suivant:

I_0

E::=.E$

E::=.E+T

E::=.T

T::=.T*F

T::=.F

F::=.ident

F::=.(E)

I_4
F::=ident.

I_0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)

I_4
F::=ident.

I_1
S::=E.$
S::=E.+T

I_0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)

I_4
F::=ident.

I_1
S::=E.$
S::=E.+T

I_0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)

I_5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)

I_4
F::=ident.

I_1
S::=E.$
S::=E.+T

I_0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)

I_6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)

I_5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)

I_4
F::=ident.

I_1
S::=E.$
S::=E.+T

I_0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)

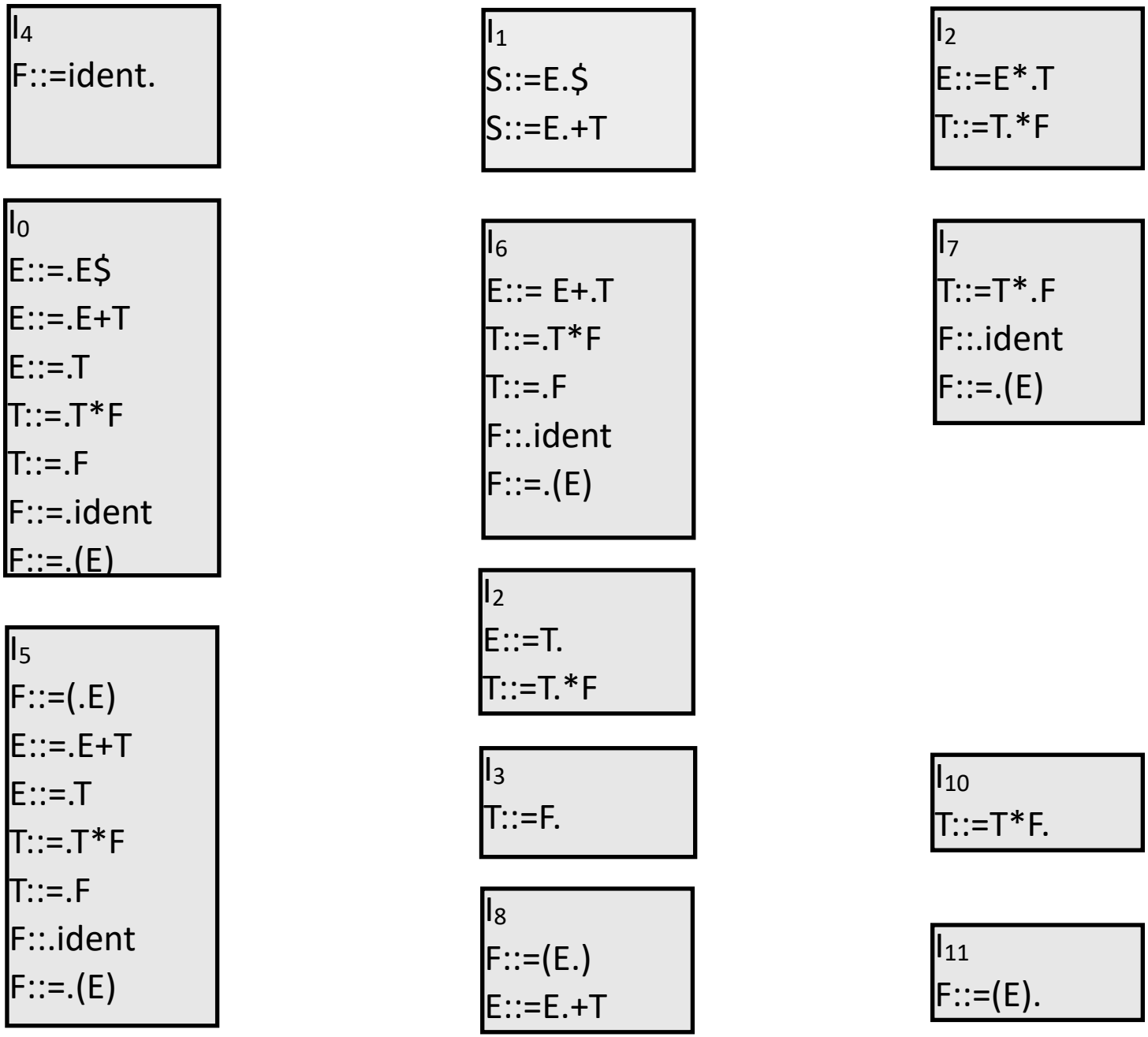
I_6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)

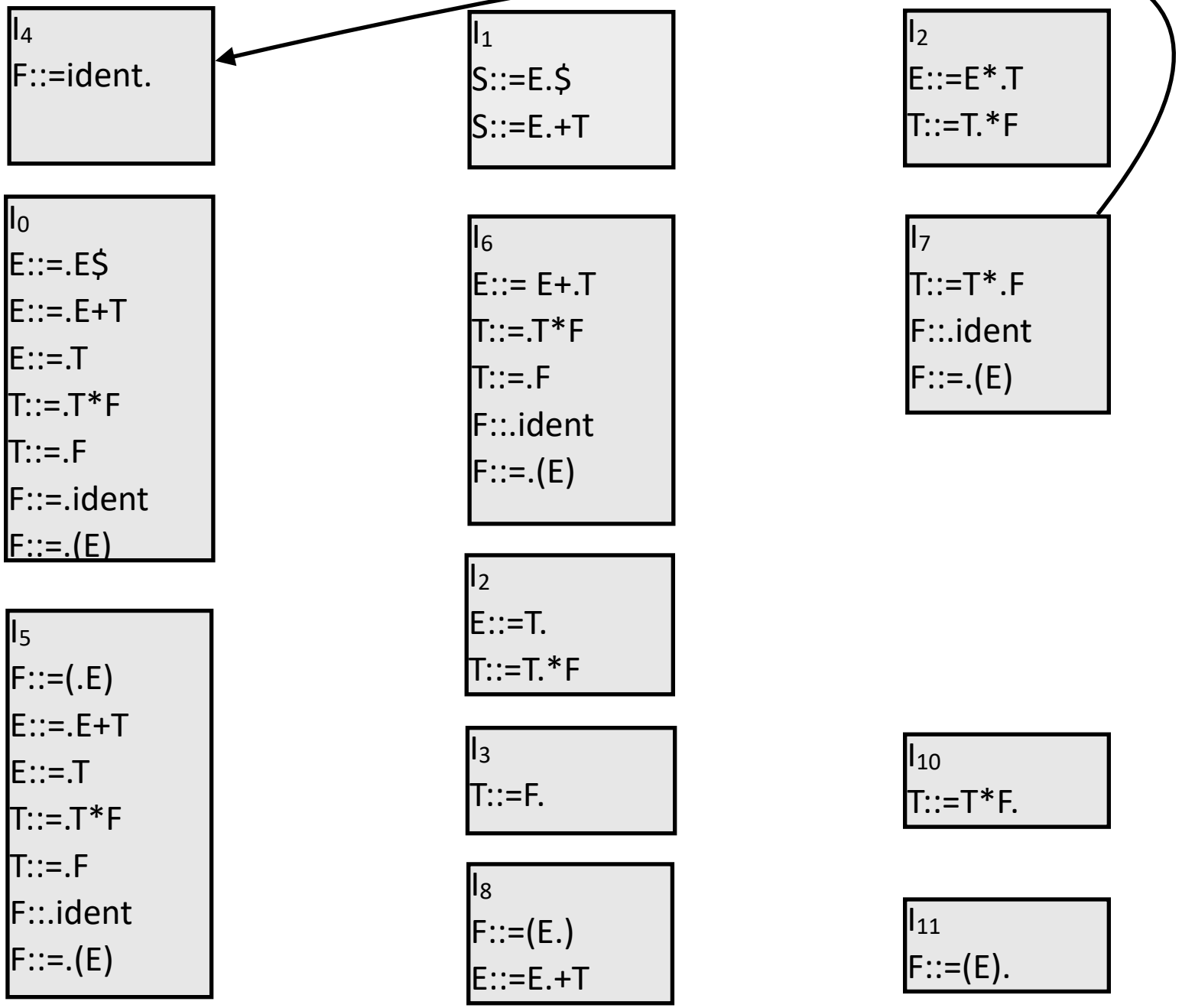
I_5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)

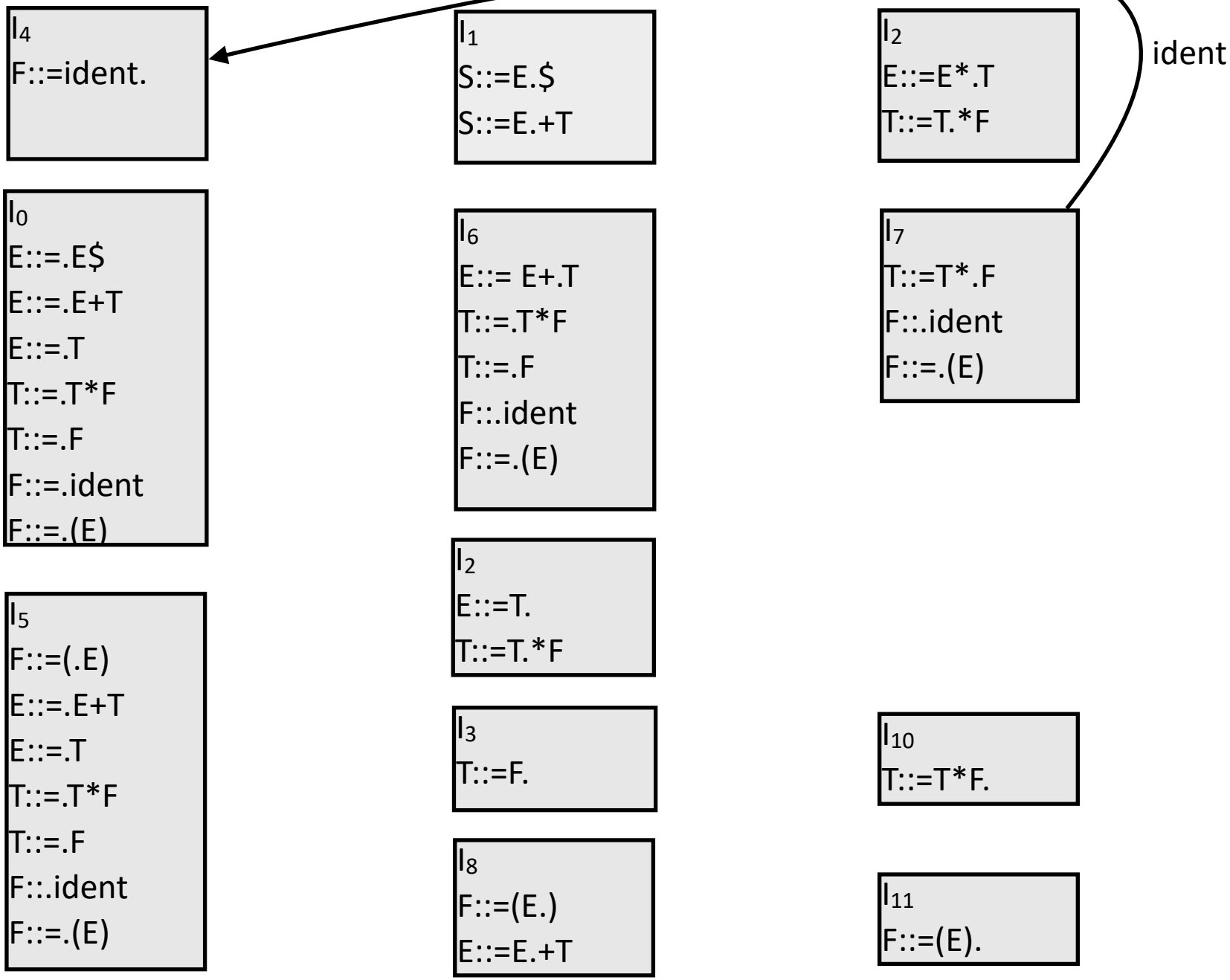
I_2
E::=T.
T::=T.*F

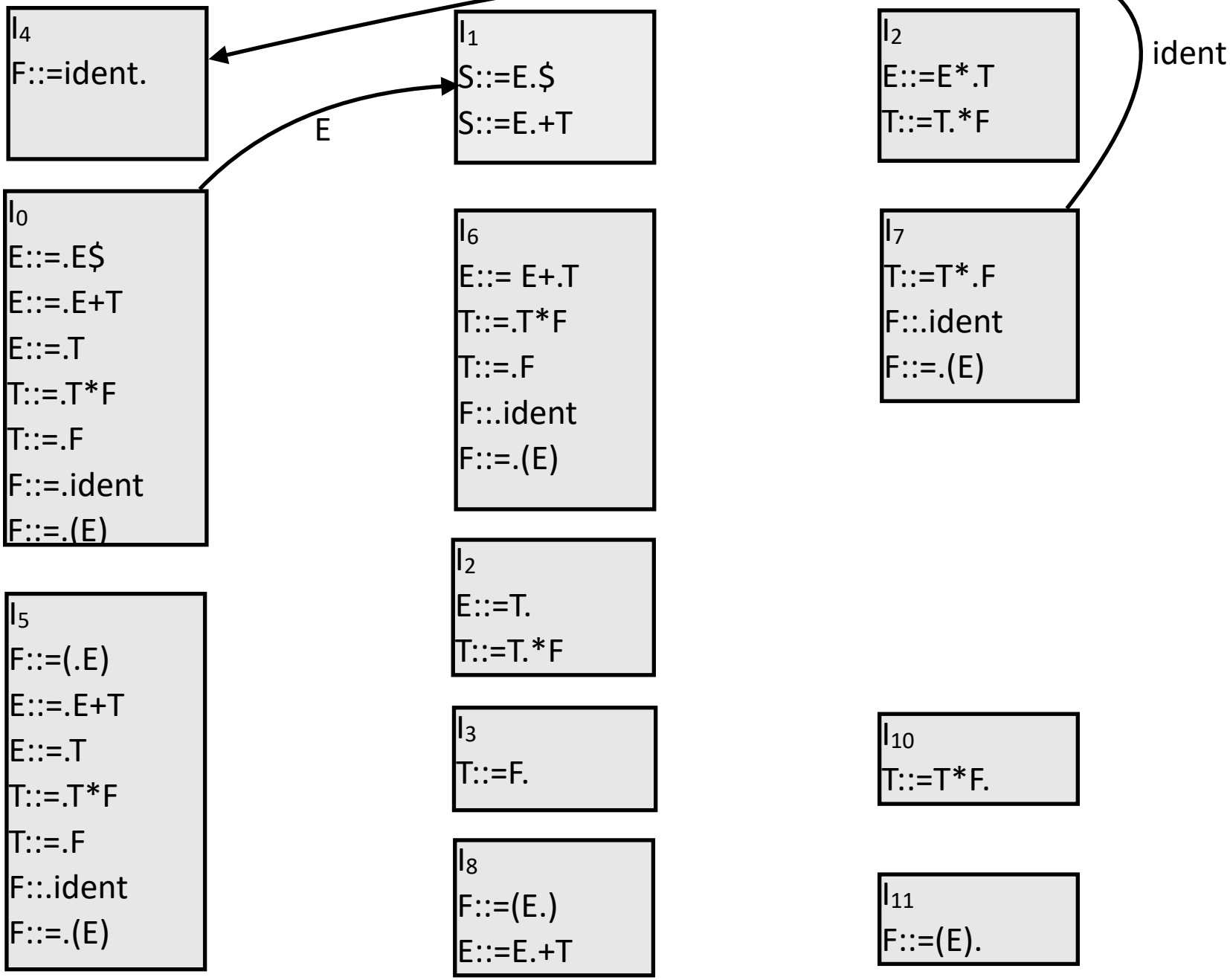
I_3
T::=F.

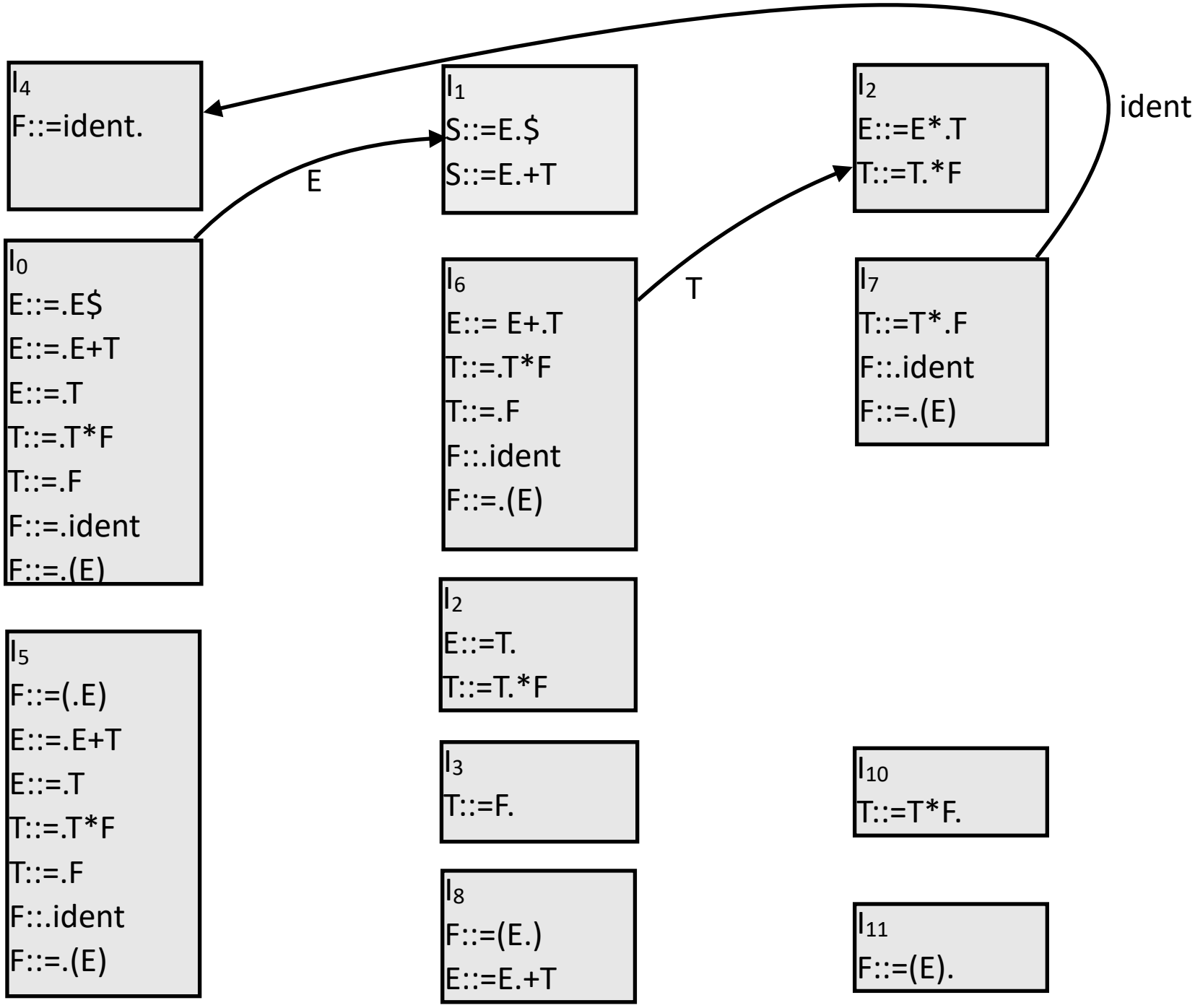
I_8
F::=(E.)
E::=E.+T

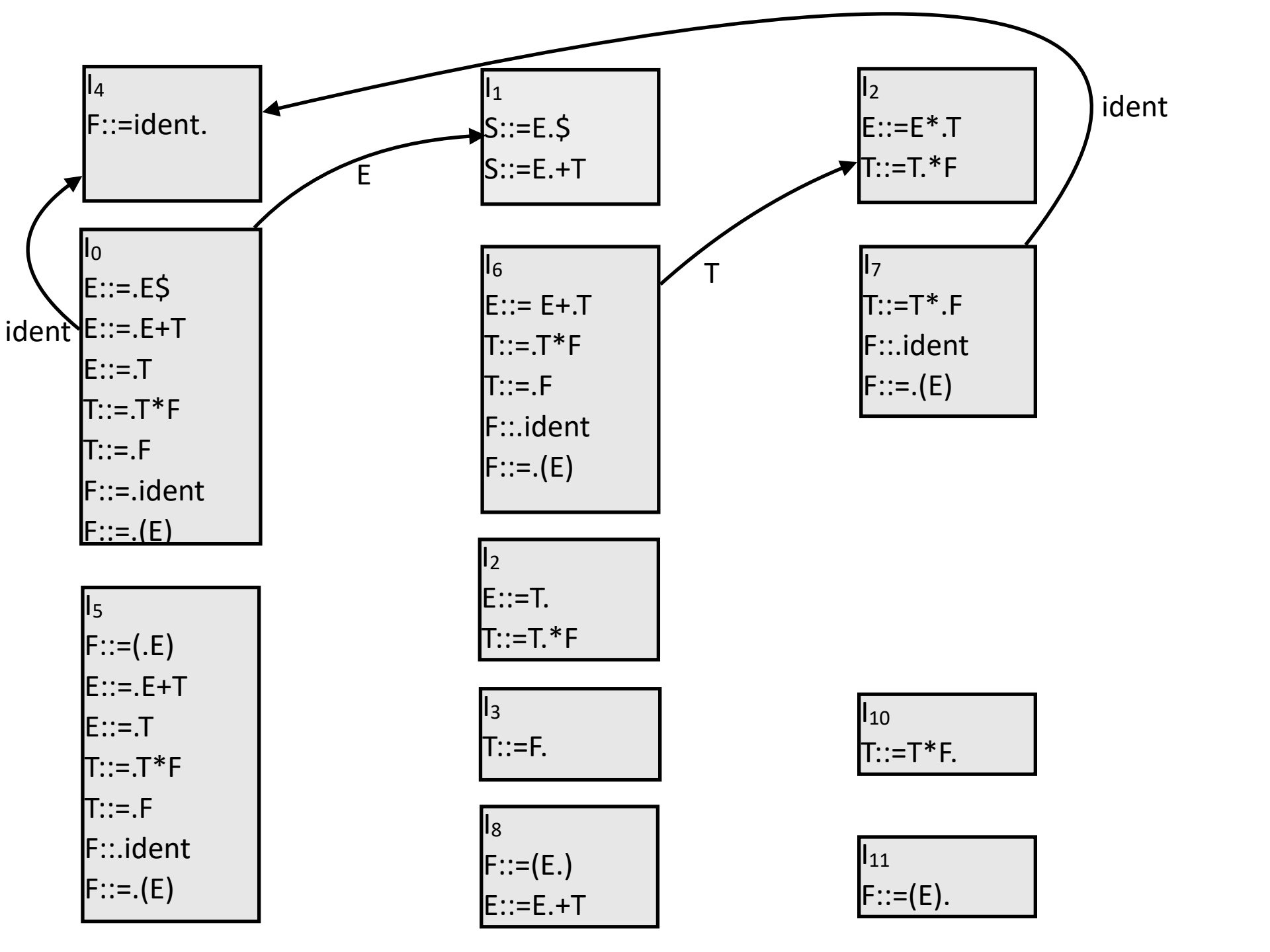

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I7
T::=T*.F
F::.ident
F::=.(E)
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

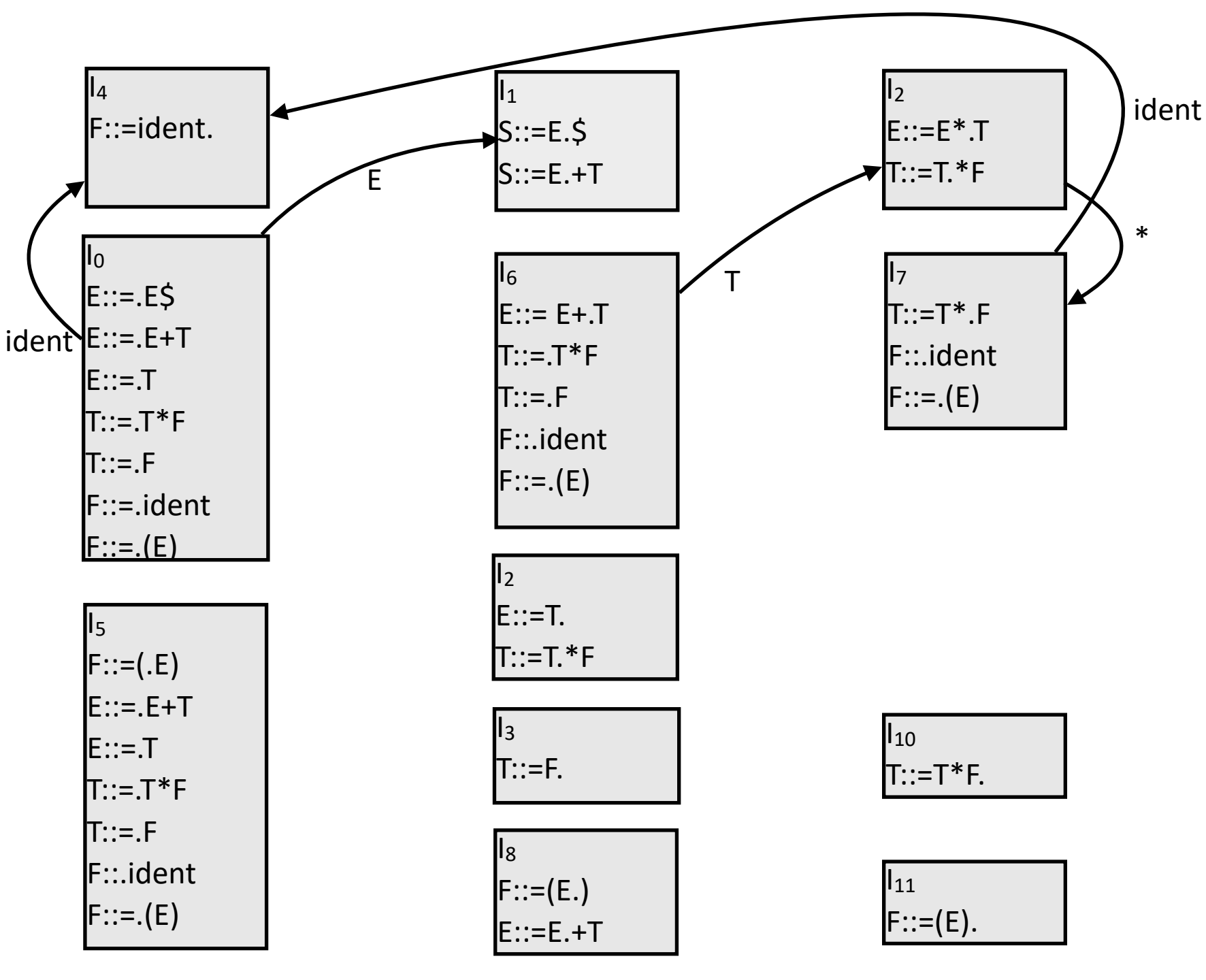

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I7
T::=T*.F
F::.ident
F::=.(E)
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

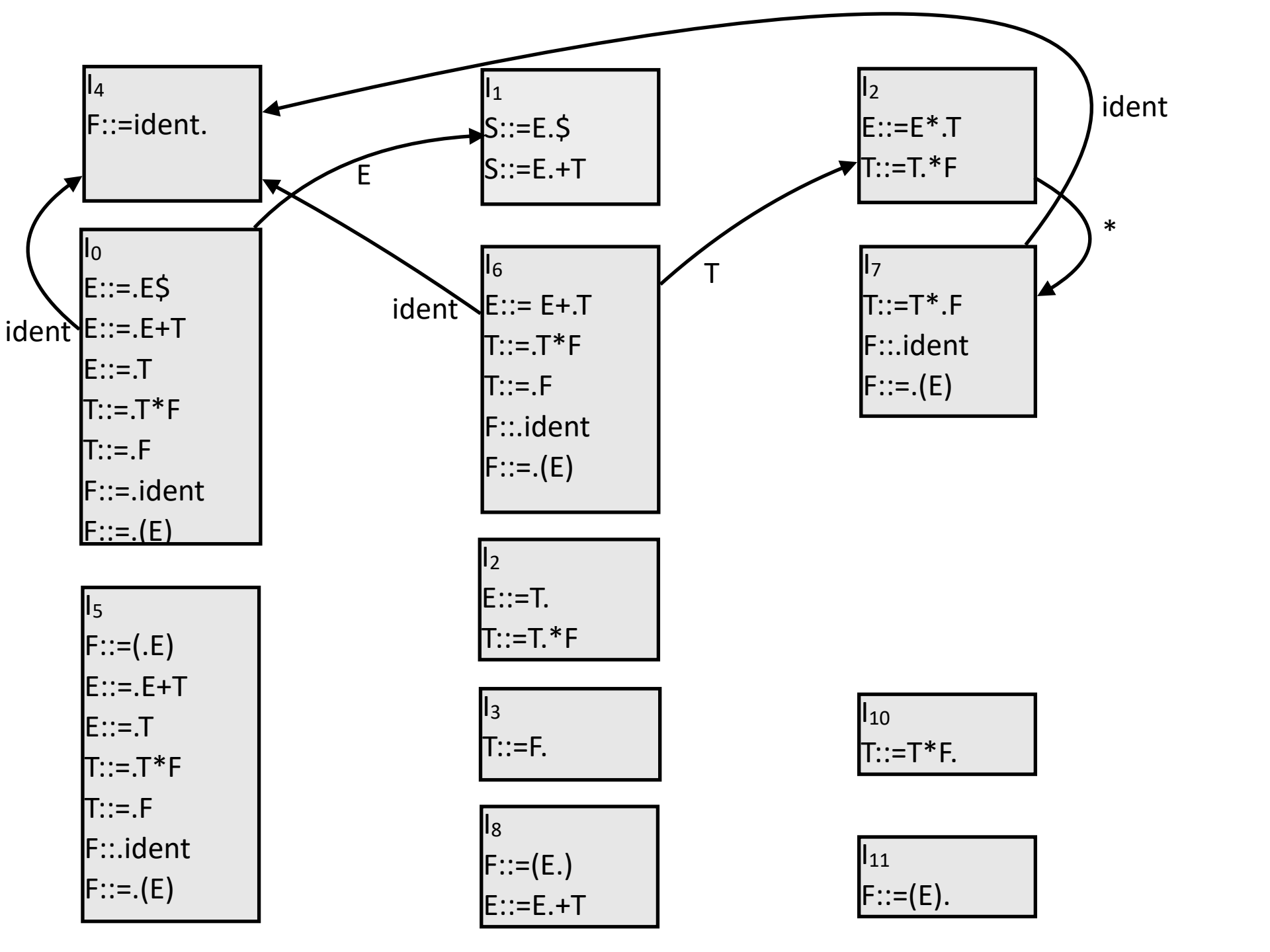

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I7
T::=T*.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

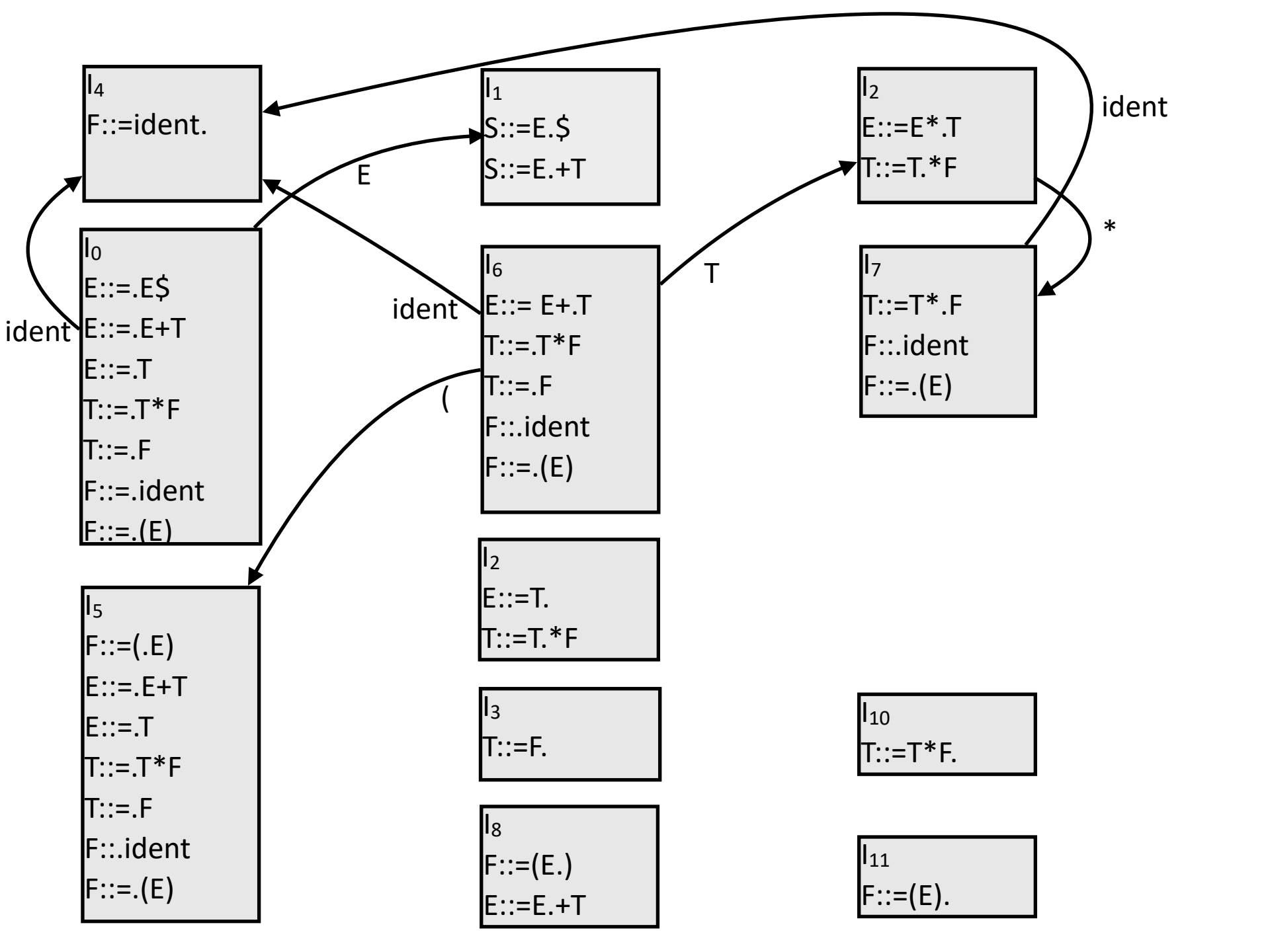

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I7
T::=T*.F
F::.ident
F::=.(E)
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

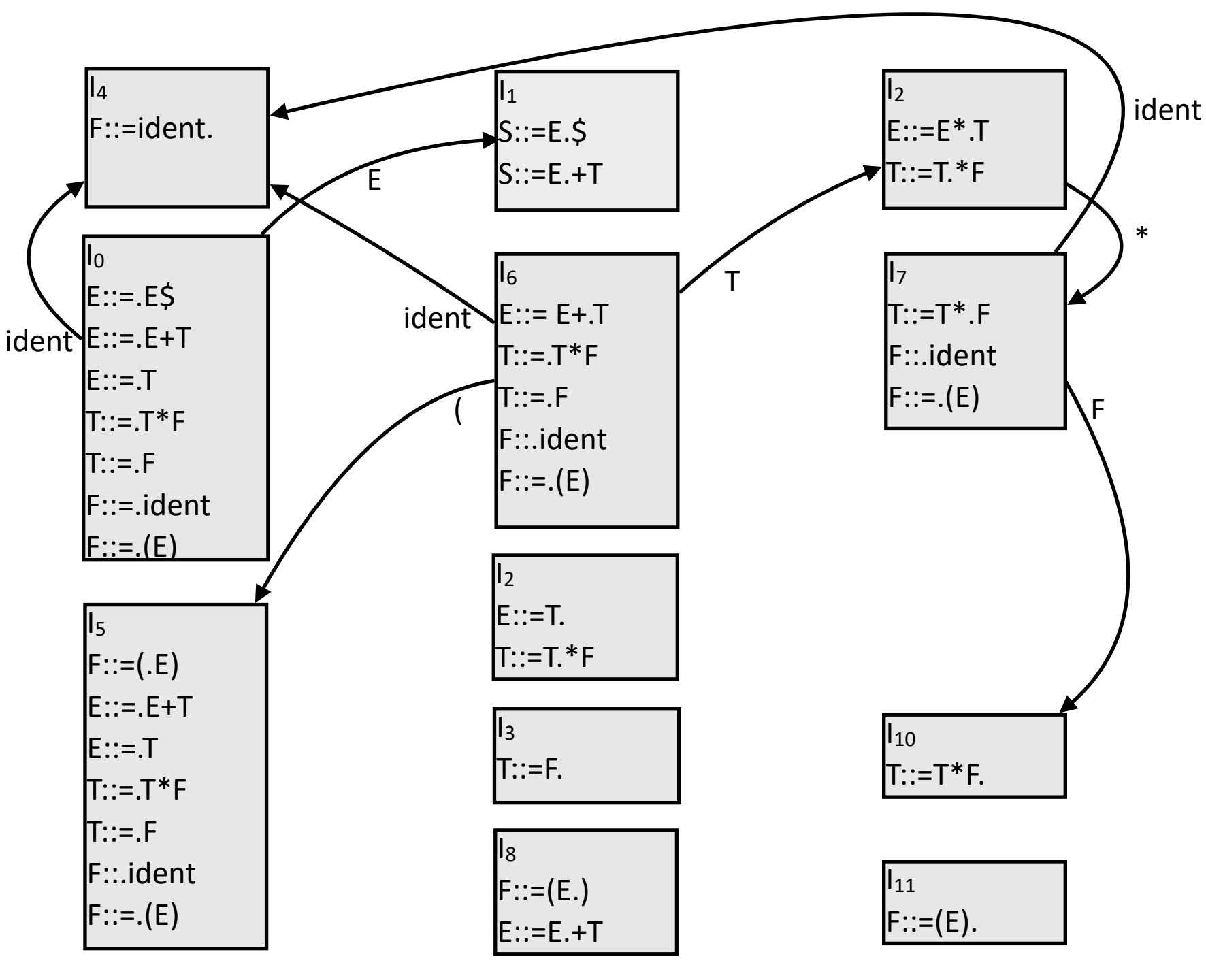

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

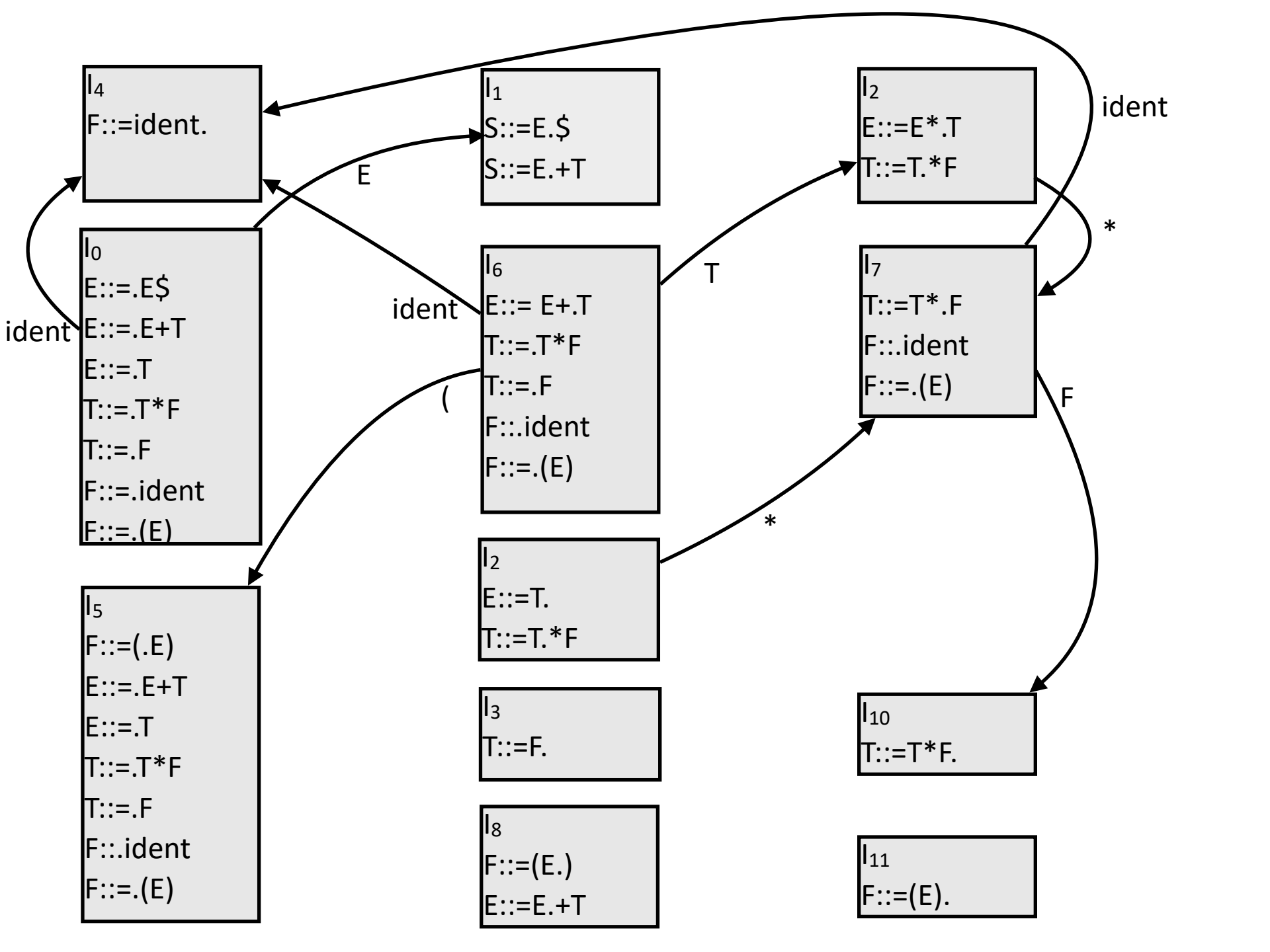

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

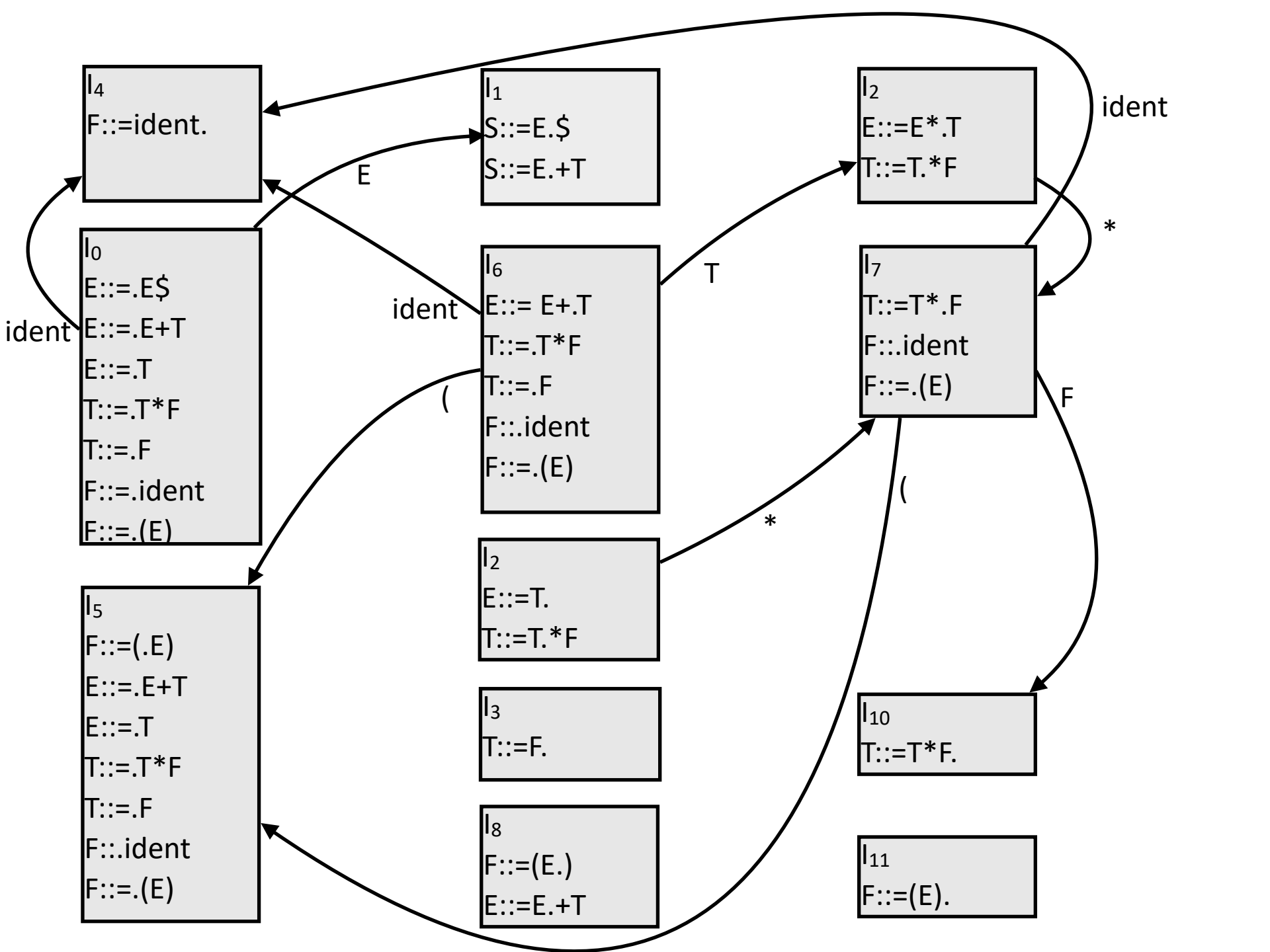

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
E
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

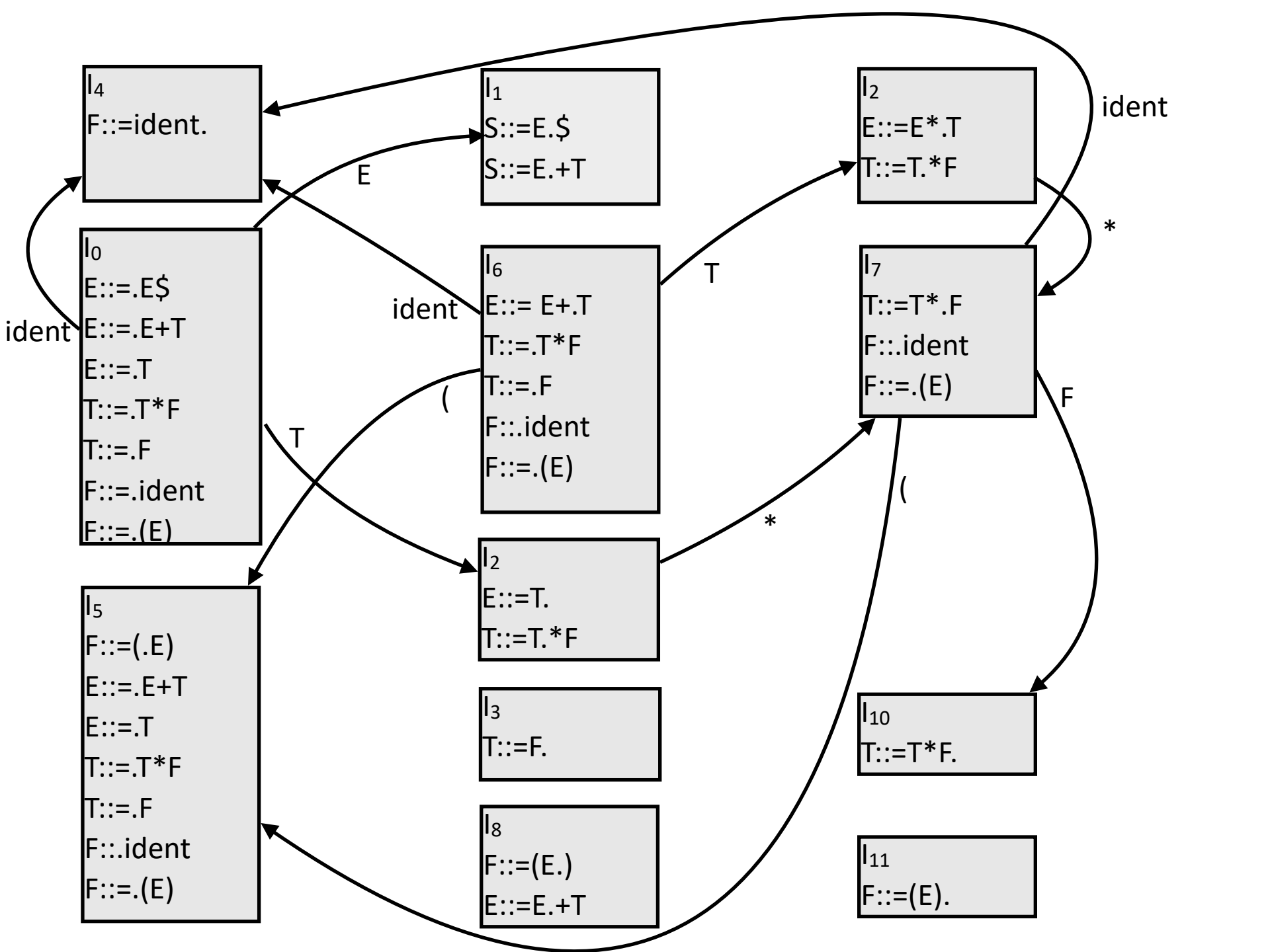

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
*
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

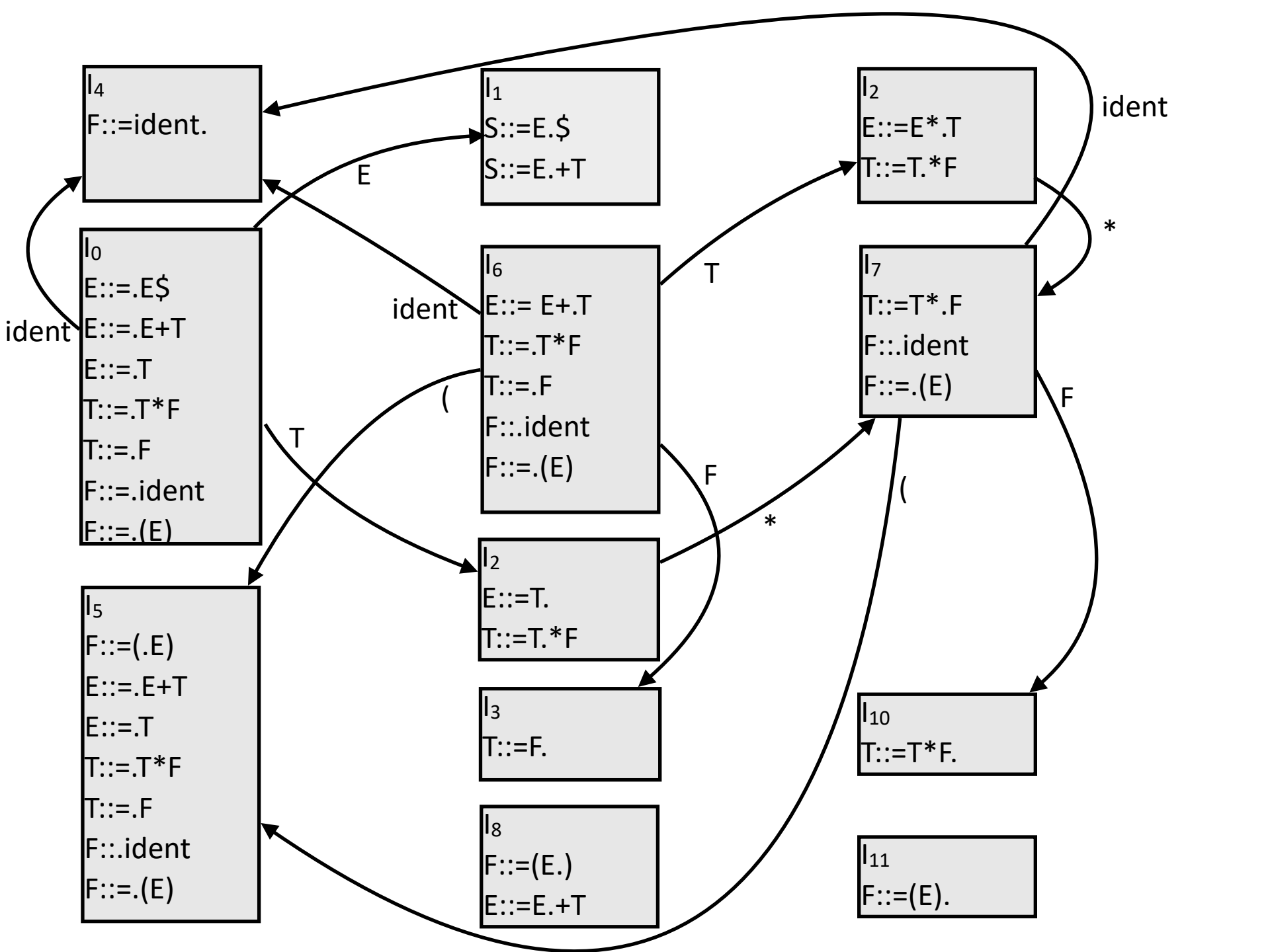

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
*
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
(
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

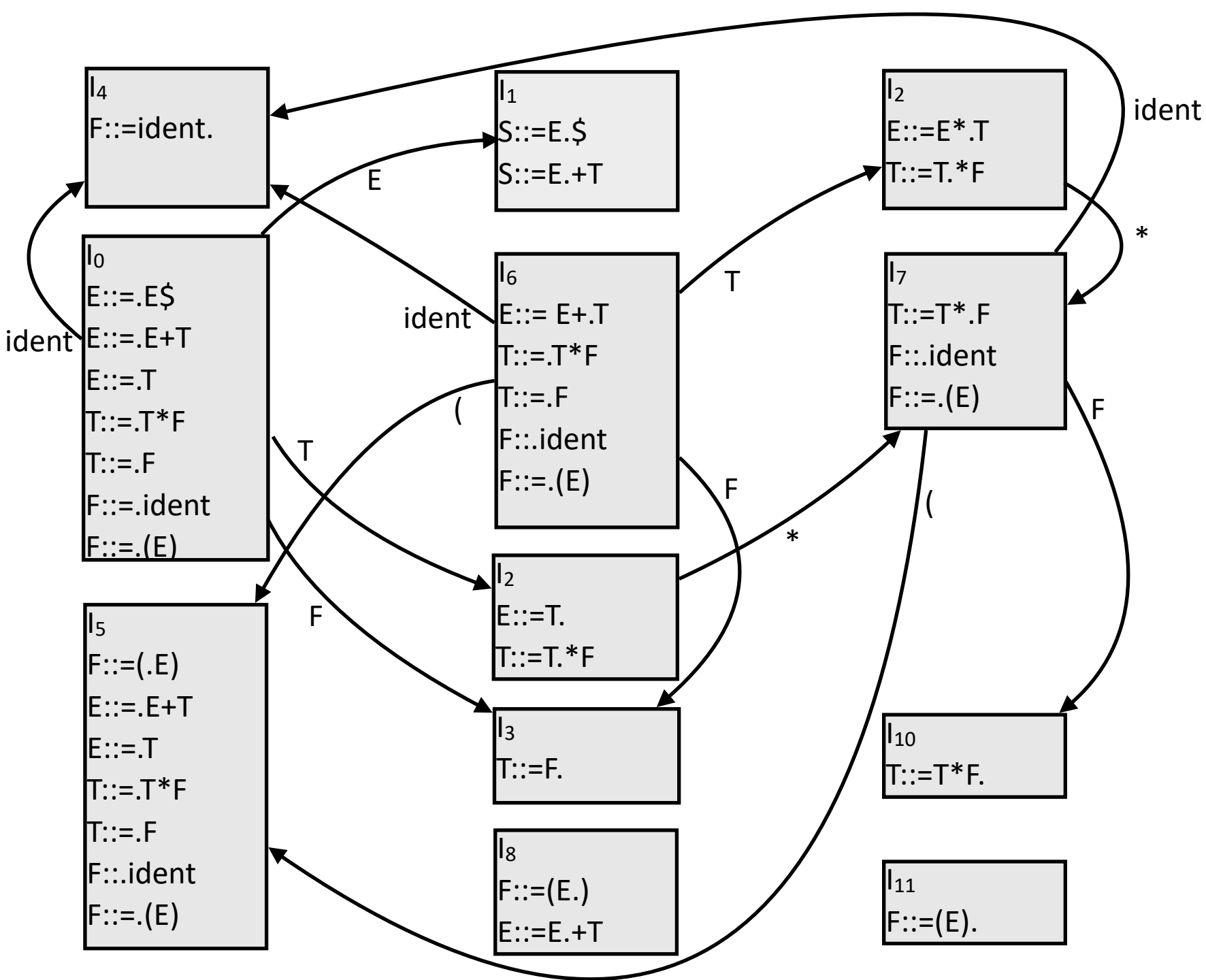

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
F
(
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I2
E::=T.
T::=T.*F
I3
T::=F.
I8
F::=(E.)
E::=E.+T
I10
T::=T*F.
I11
F::=(E).

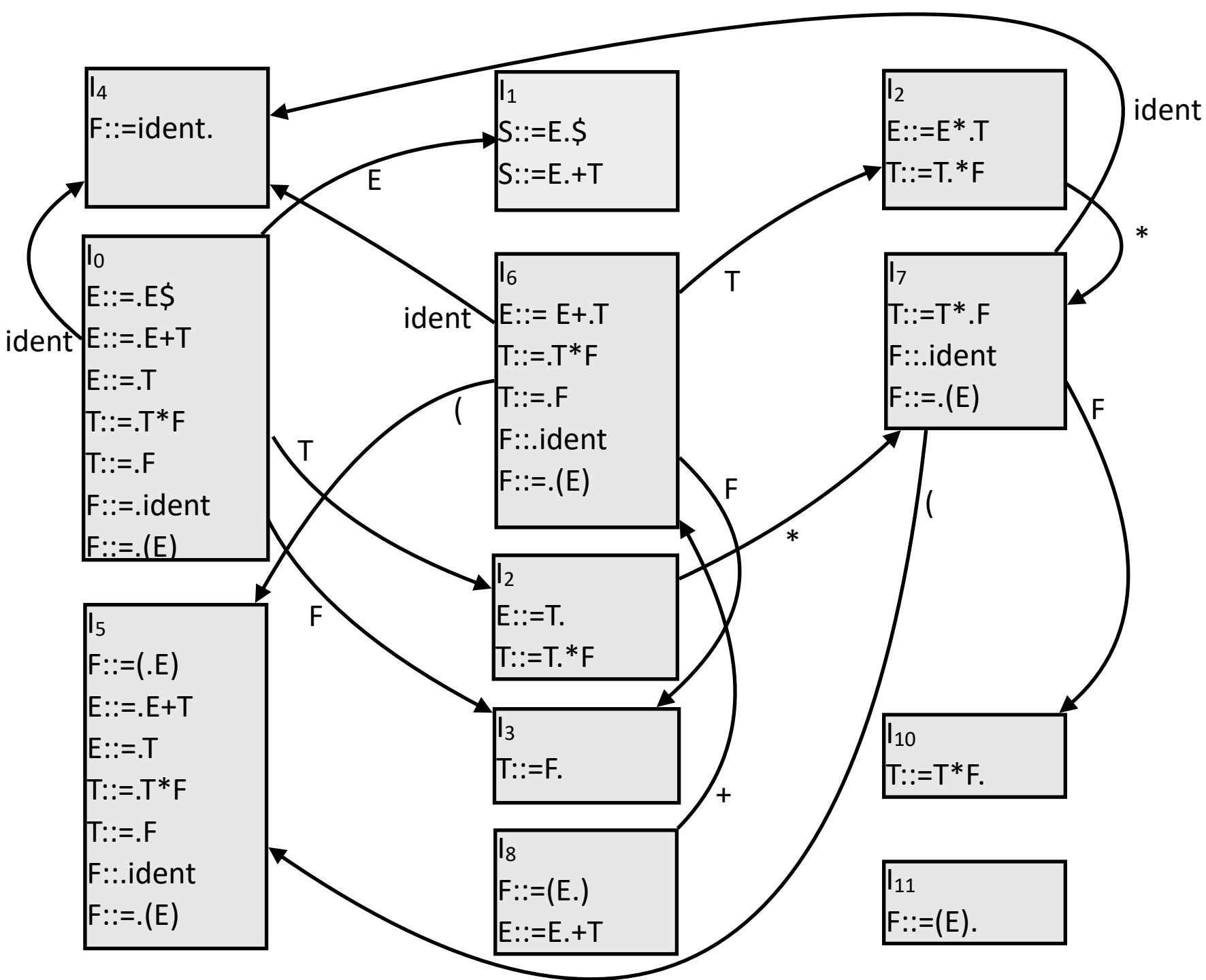

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
E
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
F
(
*
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

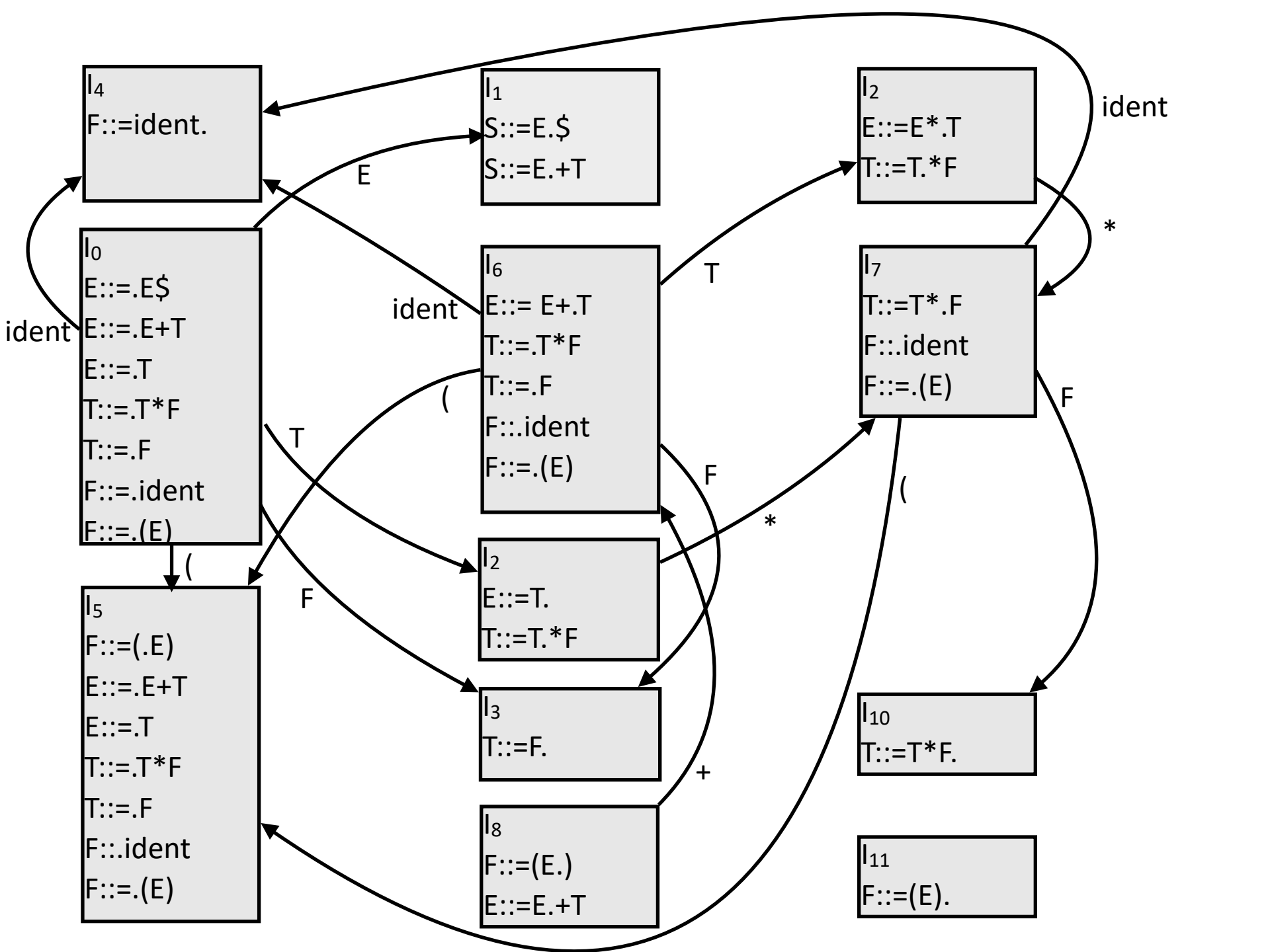

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
E
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
ident
(
I2
E::=T.
T::=T.*F
*
F
(
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

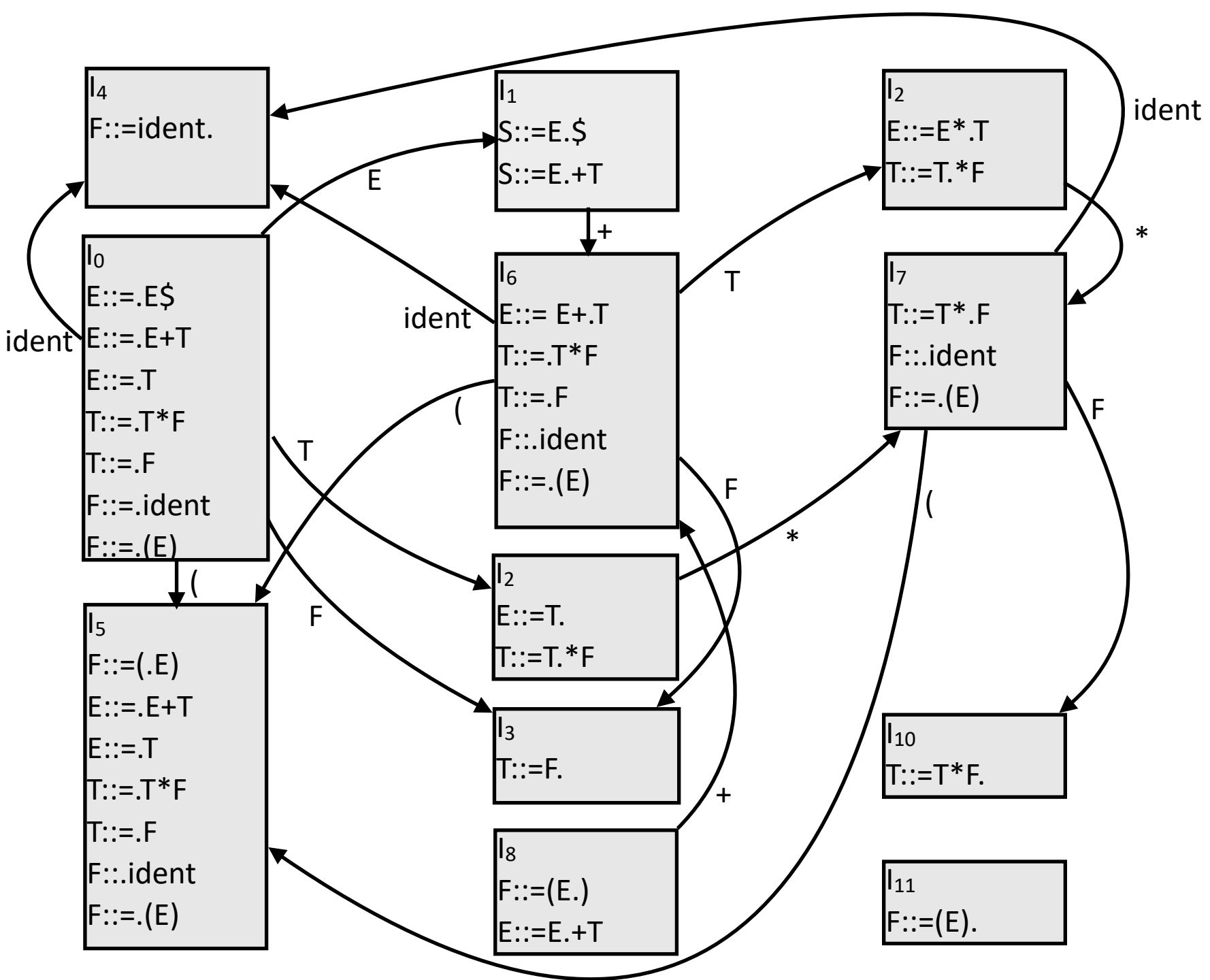

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
E
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
ident
T
(
F
I2
E::=T.
T::=T.*F
*
(
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).

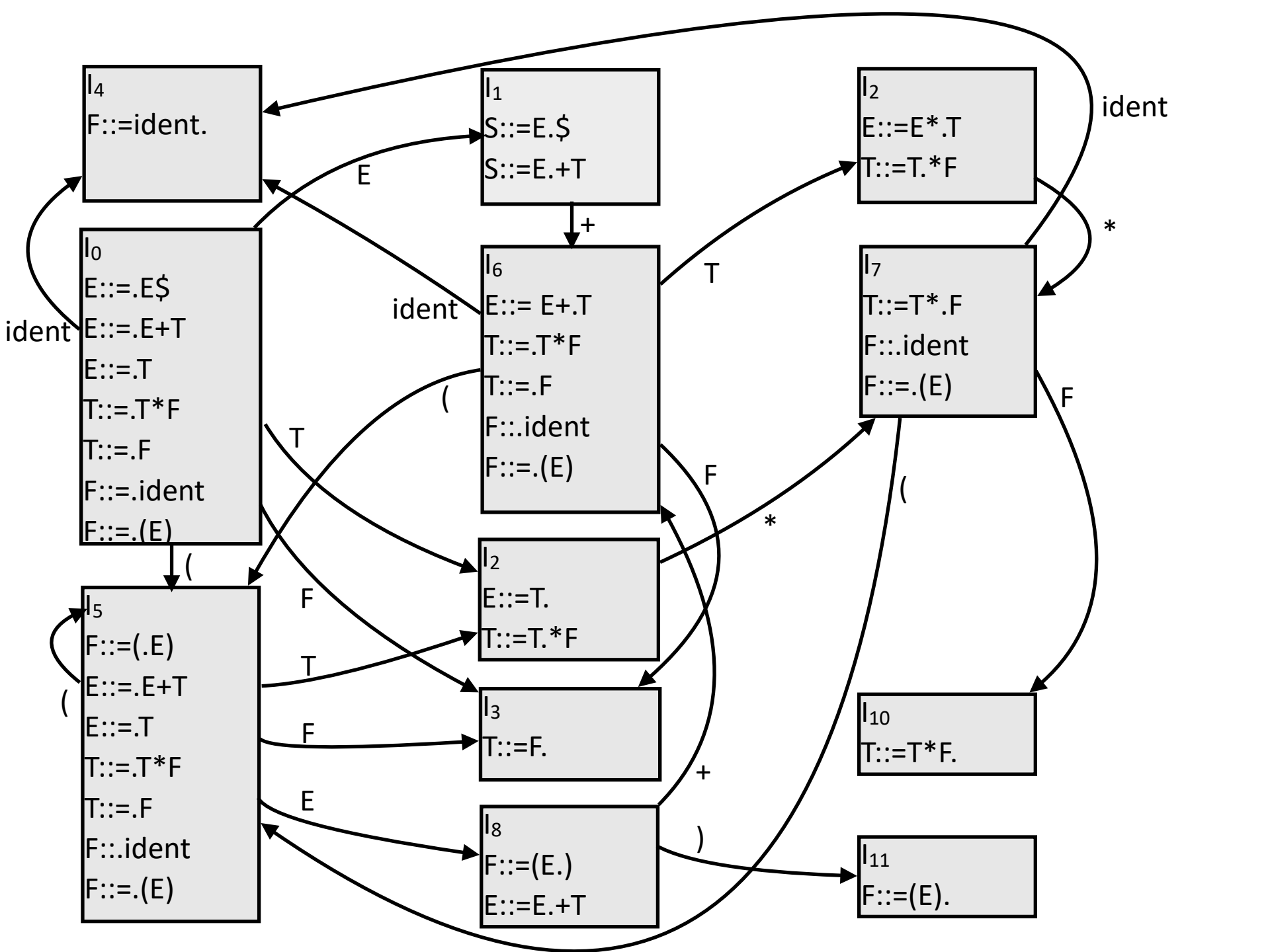

I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
E
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
ident
T
(
F
I2
E::=T.
T::=T.*F
F
*
(
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).


I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
*
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
F
(
T
F
*
(
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
F
I3
T::=F.
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).


I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
E
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
*
F
(
T
F
*
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
I3
T::=F.
+
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).


I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
*
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
E
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
ident
T
I7
T::=T*.F
F::.ident
F::=.(E)
F
(
T
F
*
(
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
(
F
I3
T::=F.
+
I10
T::=T*F.
I8
F::=(E.)
E::=E.+T
I11
F::=(E).


I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
+
*
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
(
T
F
F
*
(
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
F
I3
T::=F.
I10
T::=T*F.
+
I8
F::=(E.)
E::=E.+T
I11
F::=(E).


I4
F::=ident.
I1
S::=E.$
S::=E.+T
I2
E::=E*.T
T::=T.*F
ident
E
+
*
I0
E::=.E$
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::=.ident
F::=.(E)
ident
I6
E::= E+.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
T
I7
T::=T*.F
F::.ident
F::=.(E)
F
(
T
F
*
(
I2
E::=T.
T::=T.*F
I5
F::=(.E)
E::=.E+T
E::=.T
T::=.T*F
T::=.F
F::.ident
F::=.(E)
(
T
F
E
I3
T::=F.
I10
T::=T*F.
+
I8
F::=(E.)
E::=E.+T
)
I11
F::=(E).

SLR(1)

SLR(1)

- **Définition** : une grammaire est dite **SLR(1)** si pour tout état I de l'automate LR(0) les deux propriétés suivantes sont vérifiées :

SLR(1)

- **Définition** : une grammaire est dite **SLR(1)** si pour tout état I de l'automate LR(0) les deux propriétés suivantes sont vérifiées :

  - Si I contient un couple d'items $[\mathtt{X}: \alpha.a\beta]$ et $[\mathtt{Y}: \gamma .]$, alors $a \notin Follow(\mathtt{Y})$

SLR(1)

- **Définition** : une grammaire est dite **SLR(1)** si pour tout état I de l'automate LR(0) les deux propriétés suivantes sont vérifiées :

  - Si I contient un couple d'items [X: $\alpha.a\beta$] et [Y: γ .], alors $a \notin Follow(Y)$

  - Si I contient un couple d'items [Y: γ .] et [Z: δ .], alors $Follow(Y) \cap Follow(Z) = \varnothing$

Table d’Actions

Table d'Actions

- **Définition**: une **action** est soit S (shift) ou R (reduce):

Table d'Actions

- **Définition**: une **action** est soit S (shift) ou R (reduce):
- **Définition** : la **table d'actions SLR** est un tableau A indexé par les états de l'automate LR(0) et les symboles de $V_t \cup V_N$. On définit A[I, a], où I est un état et $a \in V_t \cup V_N$:

Table d'Actions

- **Définition**: une **action** est soit S (shift) ou R (reduce):
- **Définition** : la **table d'actions SLR** est un tableau A indexé par les états de l'automate LR(0) et les symboles de $V_t \cup V_N$. On définit A[I, a], où I est un état et $a \in V_t \cup V_N$:
  - s'il existe dans I un item [X: α.aβ], A[I, a] contient (S, J), où J est l'état cible de la transition étiquetée par a à partir de I

Table d'Actions

- **Définition**: une **action** est soit S (shift) ou R (reduce):
- **Définition** : la **table d'actions SLR** est un tableau A indexé par les états de l'automate LR(0) et les symboles de $V_t \cup V_N$. On définit A[I, a], où I est un état et $a \in V_t \cup V_N$:
  - s'il existe dans I un item [X: α.aβ], A[I, a] contient (S, J), où J est l'état cible de la transition étiquetée par a à partir de I
  - pour tout item [Y: γ .] de I et a de $Follow$(Y), A[I, a] contient (R, k) où k est le numéro de la production Y ::= γ

Table d'Actions

- **Définition**: une **action** est soit S (shift) ou R (reduce):
- **Définition** : la **table d'actions SLR** est un tableau A indexé par les états de l'automate LR(0) et les symboles de $V_t \cup V_N$. On définit A[I, a], où I est un état et $a \in V_t \cup V_N$:
  - s'il existe dans I un item [X: α.aβ], A[I, a] contient (S, J), où J est l'état cible de la transition étiquetée par a à partir de I
  - pour tout item [Y: γ .] de I et a de $Follow$(Y), A[I, a] contient (R, k) où k est le numéro de la production Y ::= γ
  - si I contient l'item [S': S.$], A[I, $] = (succès).

Exemple SLR(G3)

- La table d'actions SLR pour G_3 est donnée ci-dessous en numérotant les productions comme suit :

```
S ::= E  $          1
E ::= E + T         2
   | T              3
T ::= T * F         4
   | F              5
F ::= ident         6
   | ( E )          7
```

SLR(1) Table d'actions

- Le résultat est suivant:

	ident	(	)	+	*	$	E	T	F
I_0	(S, I_4)	(S, I_5)					(S, I_1)	(S, I_2)	(S, I_3)
I_1				(S, I_6)		(succès)			
I_2			(R, 3)	(R, 3)	(S, I_7)	(R, 3)			
I_3			(R, 5)	(R, 5)	(R, 5)	(R, 5)			
I_4			(R, 6)	(R, 6)	(R, 6)	(R, 6)			
I_5	(S, I_4)	(S, I_5)					(S, I_8)	(S, I_2)	(S, I_3)
I_6	(S, I_4)	(S, I_5)						(S, I_9)	(S, I_3)
I_7	(S, I_4)	(S, I_5)							(S, I_{10})
I_8			(S, I_{11})	(S, I_6)					
I_9			(R, 2)	(R, 2)	(S, I_7)	(R, 2)			
I_{10}			(R, 4)	(R, 4)	(R, 4)	(R, 4)			
I_{11}			(R, 7)	(R, 7)	(R, 7)	(R, 7)			

Procédure d’analyse SLR(1)

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 \ldots w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 \ldots w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 \ldots w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

Pile<Etat> P ; *Etat s, z, z'; Entier i ; Char c ;*
P := *pilevide*; empiler(P, I_0) ; -- *I_0 est l'état initial de l'automate*

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

Pile<Etat> P ; *Etat s, z, z'; Entier i ; Char c ;*
P := *pilevide*; empiler(P, I_0) ; -- *I_0 est l'état initial de l'automate*
i := 1; c := *w_1;* -- *initialiser c avec la première lettre du mot*

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

Pile<Etat> P ; *Etat s, z, z'; Entier i ; Char c ;*
P := *pilevide*; empiler(P, I_0) ; -- *I_0 est l'état initial de l'automate*
i := 1; c := *w_1;* -- *initialiser c avec la première lettre du mot*
répéter

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 \ldots w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;        -- I0 est l'état initial de l'automate
i := 1; c := w1;                          -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                        -- récupérer l'état courant de l'automate
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;        -- I0 est l'état initial de l'automate
i := 1; c := w1;                          -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                        -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 \ldots w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 \ldots w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;   -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
    -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
    -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
    -- A[z, Y] est forcément de la forme (S, z') ; empiler z'.
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
    -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
    -- A[z, Y] est forcément de la forme (S, z') ; empiler z'.
    depiler| δ |(P); z := sommet(P); z' := A[z, Y]; empiler(P, z');
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
    -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
    -- A[z, Y] est forcément de la forme (S, z') ; empiler z'.
    depiler| δ |(P); z := sommet(P); z' := A[z, Y]; empiler(P, z');
    sinon si A[s, c] = (succès) renvoyer(vrai);
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
    -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
    -- A[z, Y] est forcément de la forme (S, z') ; empiler z'.
    depiler| δ |(P); z := sommet(P); z' := A[z, Y]; empiler(P, z');
    sinon si A[s, c] = (succès) renvoyer(vrai);
    sinon renvoyer(faux);
```

Procédure d'analyse SLR(1)

- Reste a construire un algorithme qui interprete la table d'actions construit sur une grammaire G pour un mot de la forme: $w_1 \ldots w_n\$$. Voila:
- le *parser* **SLR** est définie comme suit :

procédure Analyse(**A**,$w_1 \ldots w_n\$$) : bool
-- *retourne* vrai *si le mot* $w_1 ... w_n\$$ *appartient au langage engendré par la grammaire à*
-- *partir de laquelle la table d'actions a été construite, et* faux *sinon.*
-- **A** est la table d'actions de l'analyseur

```
Pile<Etat> P ;  Etat s, z, z'; Entier i ; Char c ;
  P := pilevide; empiler(P, I0 ) ;          -- I0 est l'état initial de l'automate
i := 1; c := w1;                            -- initialiser c avec la première lettre du mot
répéter
   s := sommet(P);                          -- récupérer l'état courant de l'automate
   si A[s, c] = (S, z')
     empiler(P, z'); i := i + 1; c := wi;  -- empiler le nouvel état et avancer dans le mot
   sinon si A[s, c] = (R, k)
    -- supposons que la k^ème règle est Y  : : = δ
    -- dépiler | δ | symboles de la pile. Soit z le nouveau sommet de pile, alors
    -- A[z, Y] est forcément de la forme (S, z') ; empiler z'.
    depiler| δ |(P); z := sommet(P); z' := A[z, Y]; empiler(P, z');
    sinon si A[s, c] = (succès) renvoyer(vrai);
    sinon renvoyer(faux);
  fin
```

Gestion des Conflits

Gestion des Conflits

- La construction de la table d'actions peut résulter dans dans *plusieurs* actions pour une entre A[I,a].

Gestion des Conflits

- La construction de la table d'actions peut résulter dans dans *plusieurs* actions pour une entre A[I,a].
- On distingue:

Gestion des Conflits

- La construction de la table d'actions peut résulter dans dans *plusieurs* actions pour une entre A[I,a].
- On distingue:
    - un shift/shift conflict
      (plusieurs continuations possible)

Gestion des Conflits

- La construction de la table d'actions peut résulter dans dans *plusieurs* actions pour une entre A[I,a].
- On distingue:
    - un shift/shift conflict
      (plusieurs continuations possible)
    - un shift/reduce conflict
      (plusieurs continuations possible)

Gestion des Conflits

- La construction de la table d'actions peut résulter dans dans *plusieurs* actions pour une entre A[I,a].
- On distingue:
    - un shift/shift conflict
      (plusieurs continuations possible)
    - un shift/reduce conflict
      (plusieurs continuations possible)
    - un reduce/reduce conflict

Gestion des Conflits

- La construction de la table d'actions peut résulter dans dans *plusieurs* actions pour une entre A[I,a].
- On distingue:
    - un shift/shift conflict
      (plusieurs continuations possible)
    - un shift/reduce conflict
      (plusieurs continuations possible)
    - un reduce/reduce conflict
- Il faut disambiguer la grammaire.
  Certains générateurs des parlers offrent aussi la possibilité de spécifier une priorité.

Exemple: Reduce/reduce conflict

```
S ::= A a A b | B b B a
A ::= ε
B ::= ε
```

Exemple: Reduce/reduce conflict

- On considère la grammaire:

```
S ::= A a A b | B b B a
A ::= ε
B ::= ε
```

Exemple: Reduce/reduce conflict

- On considère la grammaire:

```
S ::= A a A b | B b B a
A ::= ε
B ::= ε
```

- Il y a un conflit reduce/reduce entre `A ::= ε` et `B ::= ε`. Pourtant, ce conflit est bénin:
  - si le premier symbole est a, seulement `A ::= ε` est possible
  - si le premier symbole est b, seulement `B ::= ε` est possible.

Exemple: Reduce/reduce conflict

- On considère la grammaire:

```
S ::= A a A b | B b B a
A ::= ε
B ::= ε
```

- Il y a un conflit reduce/reduce entre `A ::= ε` et `B ::= ε`. Pourtant, ce conflit est bénin:
  - si le premier symbole est a, seulement `A ::= ε` est possible
  - si le premier symbole est b, seulement `B ::= ε` est possible.

- La construction du automate LR(1) est toujours possible étant donné quelle est basé sur les First est Follow sets qui sont disjoint.

Generation des Parsers

Generation des Parsers

- Important prerequisite:
  Déterminer si les conditions de déterminisme sont satisfaites.

Generation des Parsers

- Important prerequisite:
  Déterminer si les conditions de déterminisme sont satisfaites.
- Beaucoup des définitions des langages viennent avec une grammaire LR(1) de nos jours, desfois avec des priorités

Generation des Parsers

- Important prerequisite:
  Déterminer si les conditions de déterminisme sont satisfaites.
- Beaucoup des définitions des langages viennent avec une grammaire LR(1) de nos jours, desfois avec des priorités
- Pas discuté en détail: construction d'une AST

Generation des Parsers

- Important prerequisite:
  Déterminer si les conditions de déterminisme sont satisfaites.
- Beaucoup des définitions des langages viennent avec une grammaire LR(1) de nos jours, desfois avec des priorités
- Pas discuté en détail: construction d'une AST
- Pas discuté en détail: traitement des erreurs. (stratégies de dépiler jusqu'on arrive a des parties reconnaissables: catch-sets)

Generation des Parsers

- Important prerequisite:
  Déterminer si les conditions de déterminisme sont satisfaites.
- Beaucoup des définitions des langages viennent avec une grammaire LR(1) de nos jours, desfois avec des priorités
- Pas discuté en détail: construction d'une AST
- Pas discuté en détail: traitement des erreurs. (stratégies de dépiler jusqu'on arrive a des parties reconnaissables: catch-sets)
- Pas discuté en détail: interface entre générateur du lexer et du générateur du parser.

Generateurs

Generateurs

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:
    - Monde C, C++ : Flex + Bison

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:
  - Monde C, C++ : Flex + Bison
  - Monde Java : JLex + JavaCC + JJTree

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:
  - Monde C, C++ : Flex + Bison
  - Monde Java : JLex + JavaCC + JJTree
  - Monde OCaml : OCamlLex + Menhir (LR (1))

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:
  - Monde C, C++ : Flex + Bison
  - Monde Java : JLex + JavaCC + JJTree
  - Monde OCaml : OCamlLex + Menhir (LR (1))
  - Monde SML : MLLex + MLYacc

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:
  - Monde C, C++ : Flex + Bison
  - Monde Java : JLex + JavaCC + JJTree
  - Monde OCaml : OCamlLex + Menhir (LR (1))
  - Monde SML : MLLex + MLYacc
  - Monde Haskell : Cabal+Happy

Generateurs

- Les familles des générateurs de lexers et parser sont orienté vers des langages pour lesquelles ils génèrent le code:
  - Monde C, C++ : Flex + Bison
  - Monde Java : JLex + JavaCC + JJTree
  - Monde OCaml : OCamlLex + Menhir (LR (1))
  - Monde SML : MLLex + MLYacc
  - Monde Haskell : Cabal+Happy
  - …