Vers des analyseurs syntaxiques efficaces et bien typés

François Pottier et Yann Régis-Gianas

4 janvier 2005

En bref

Le propos de cet exposé est d'illustrer le slogan informel selon lequel *un système de types expressif permet de garantir la sûreté de programmes complexes*.

La classe de programmes considérée sera celle des *analyseurs syntaxiques LR* et le système de types *une extension de ML*.

Les analyseurs LR

On aime *spécifier* un analyseur syntaxique sous forme d'une grammaire non contextuelle, typiquement au format BNF, décorée par des actions sémantiques.

On aime *implanter* un analyseur syntaxique sous forme d'un automate déterministe à pile (DPDA).

Les grammaires LR sont celles pour lesquelles une telle implantation est possible.

Les générateurs d'analyseurs LR

Il existe des outils pour *engendrer*, à partir d'une grammaire LR, un programme qui simule l'exécution de l'automate correspondant.

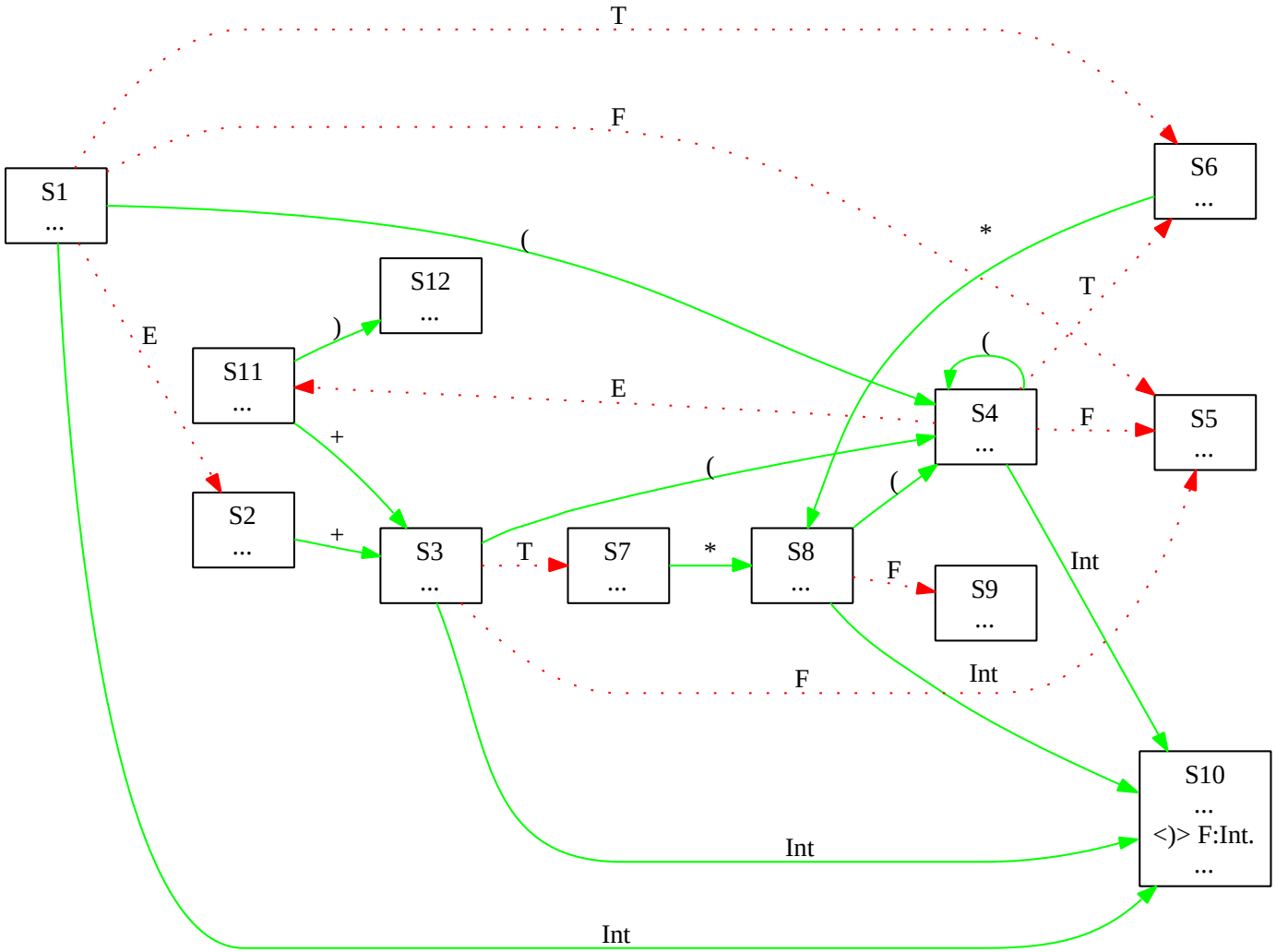
Peut-on garantir la *sûreté* des programmes ainsi engendrés sans devoir exiger une *confiance* en la correction de l'outil ?

Une grammaire simple

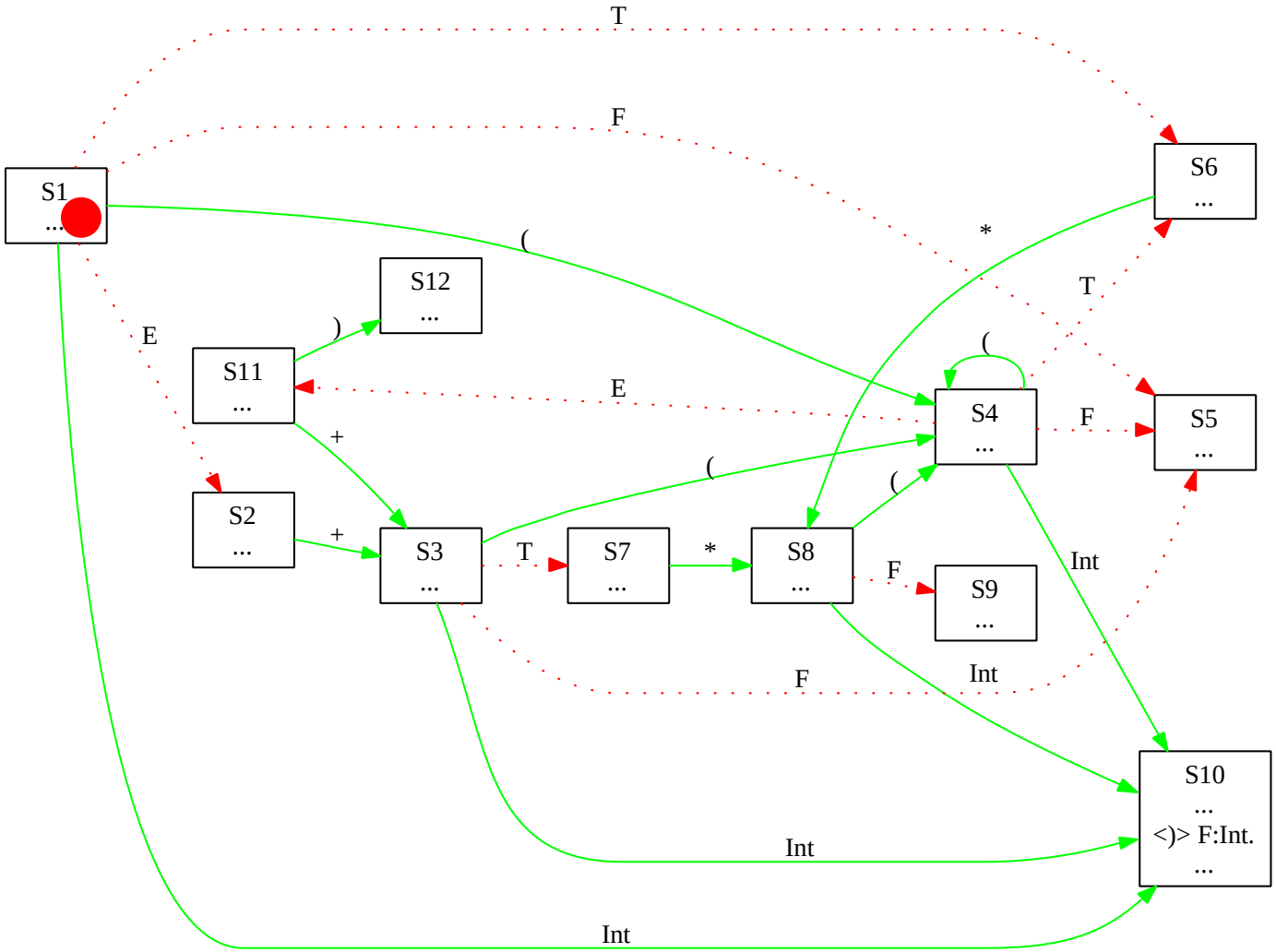
Voici une grammaire LR très simple, tirée du « Dragon Book: »

$$
\begin{array}{lrcl}
(1) & E\{x\} + T\{y\} & \longrightarrow & E\{x+y\} \\
(2) & T\{x\} & \longrightarrow & E\{x\} \\
(3) & T\{x\} * F\{y\} & \longrightarrow & T\{x \times y\} \\
(4) & F\{x\} & \longrightarrow & T\{x\} \\
(5) & (E\{x\}) & \longrightarrow & F\{x\} \\
(6) & \textbf{int}\{x\} & \longrightarrow & F\{x\}
\end{array}
$$

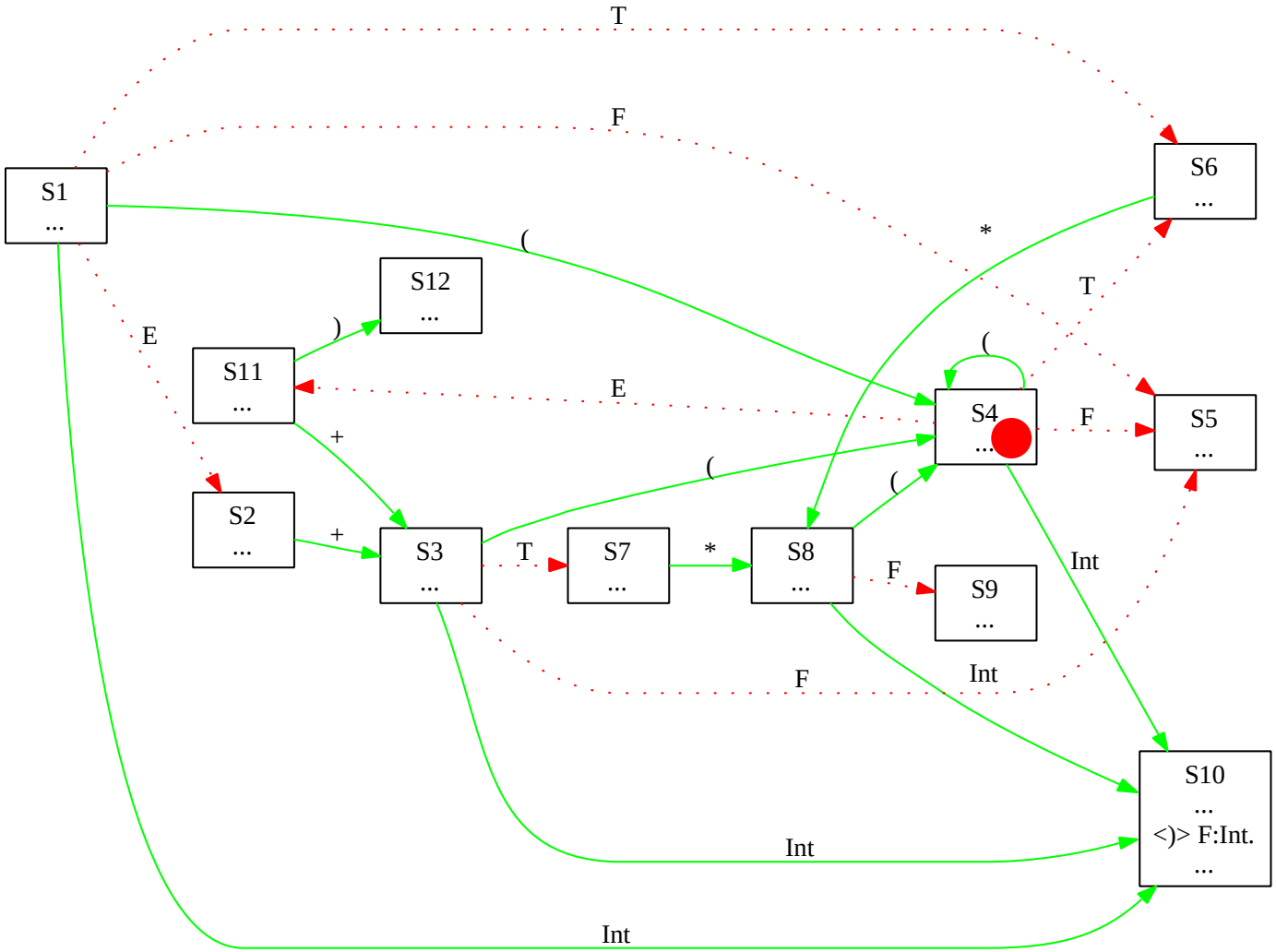
Les *terminaux* ou *lexèmes* sont **+**, *****, **(**, **)** et **int**. Les *non-terminaux* sont *E*, *T* et *F*. Les quatre premiers n'ont pas de valeur sémantique; les quatre derniers ont une valeur sémantique entière.



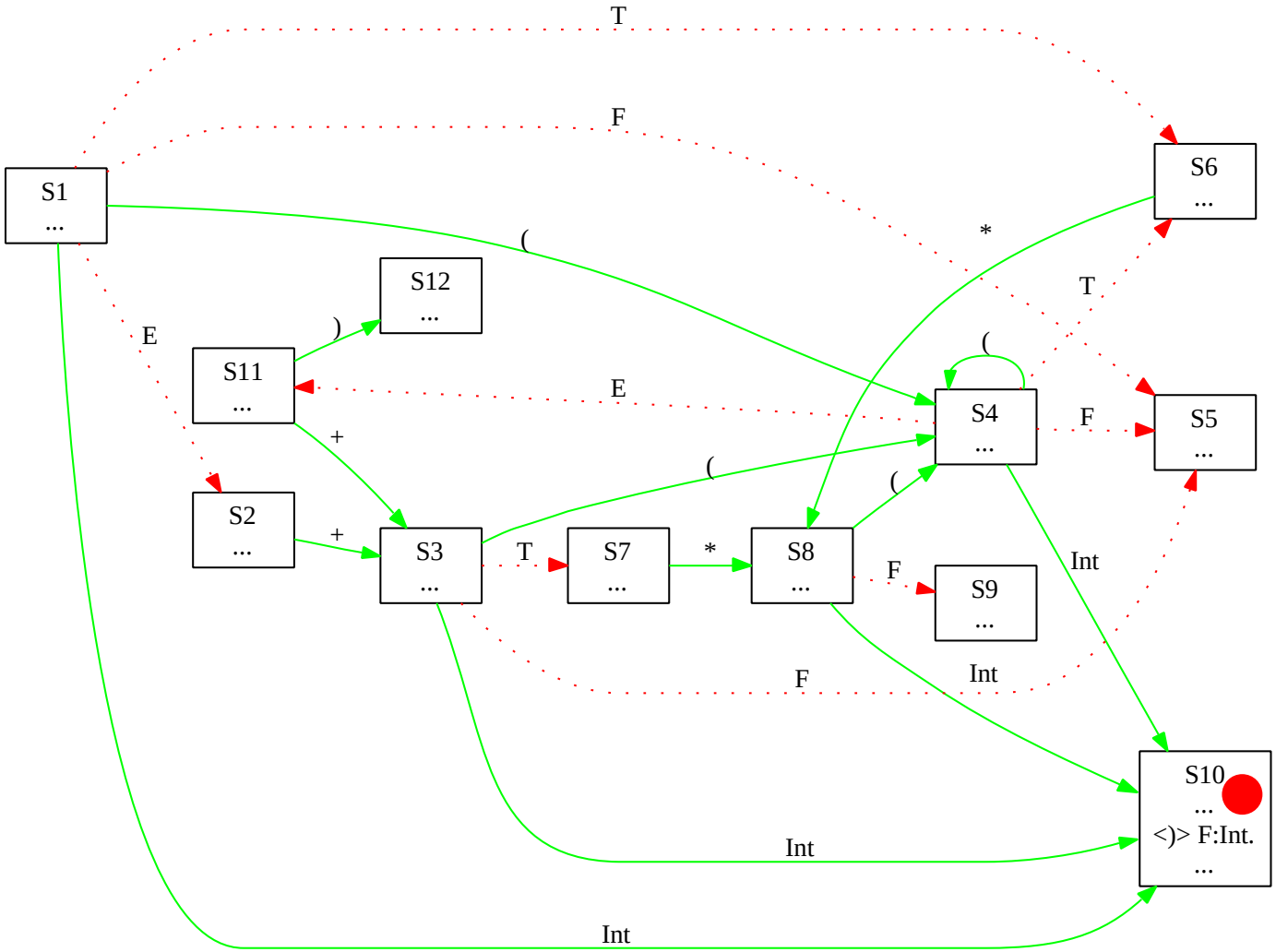
Voici un automate à pile qui reconnaît cette grammaire.



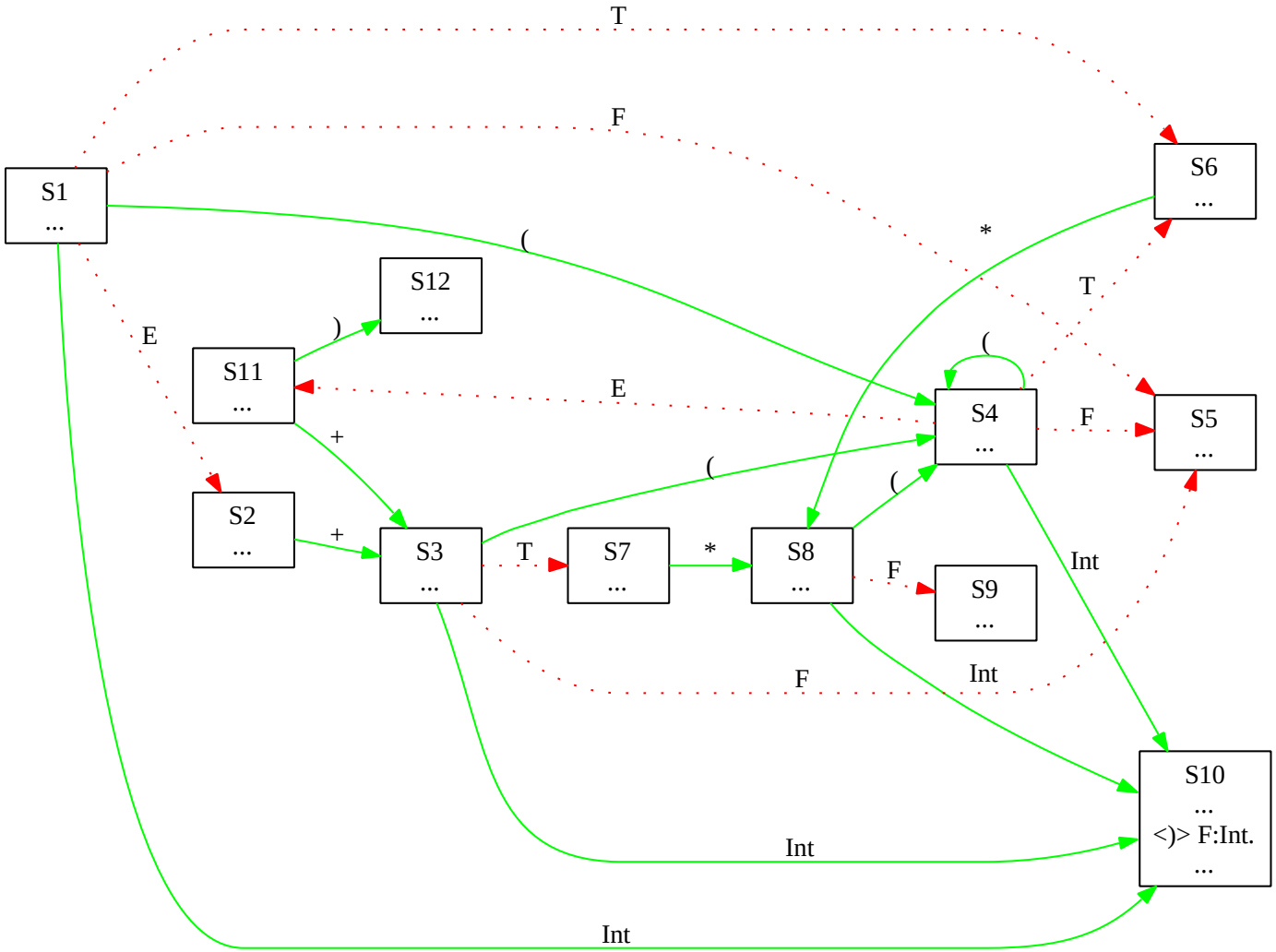
Entrée	Pile	État	Prochaine action
(**int**) $	ϵ	S_1	shift S_4



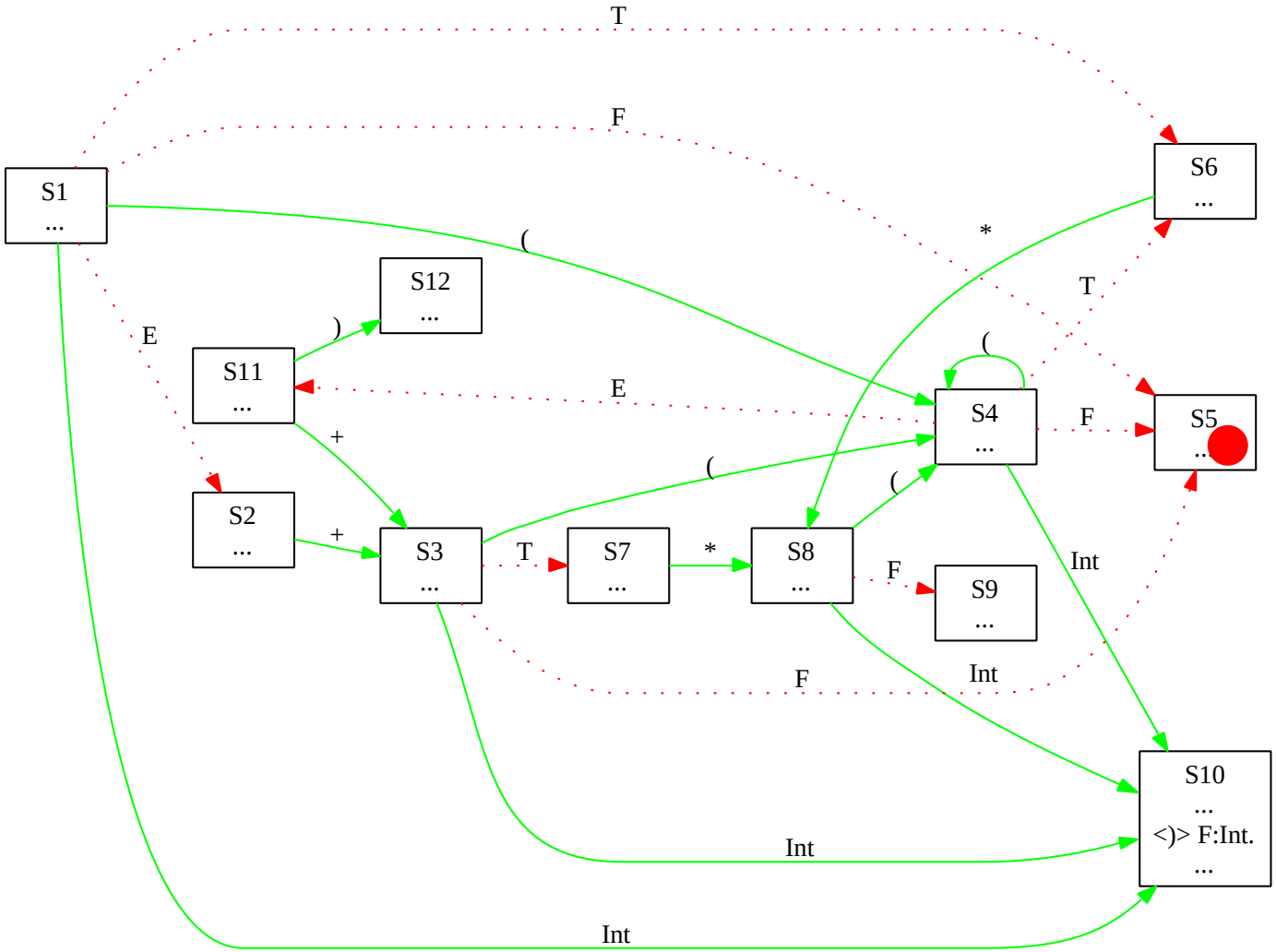
Entrée	Pile	État	Prochaine action
int) $	S_1 (	S_4	shift S_{10}



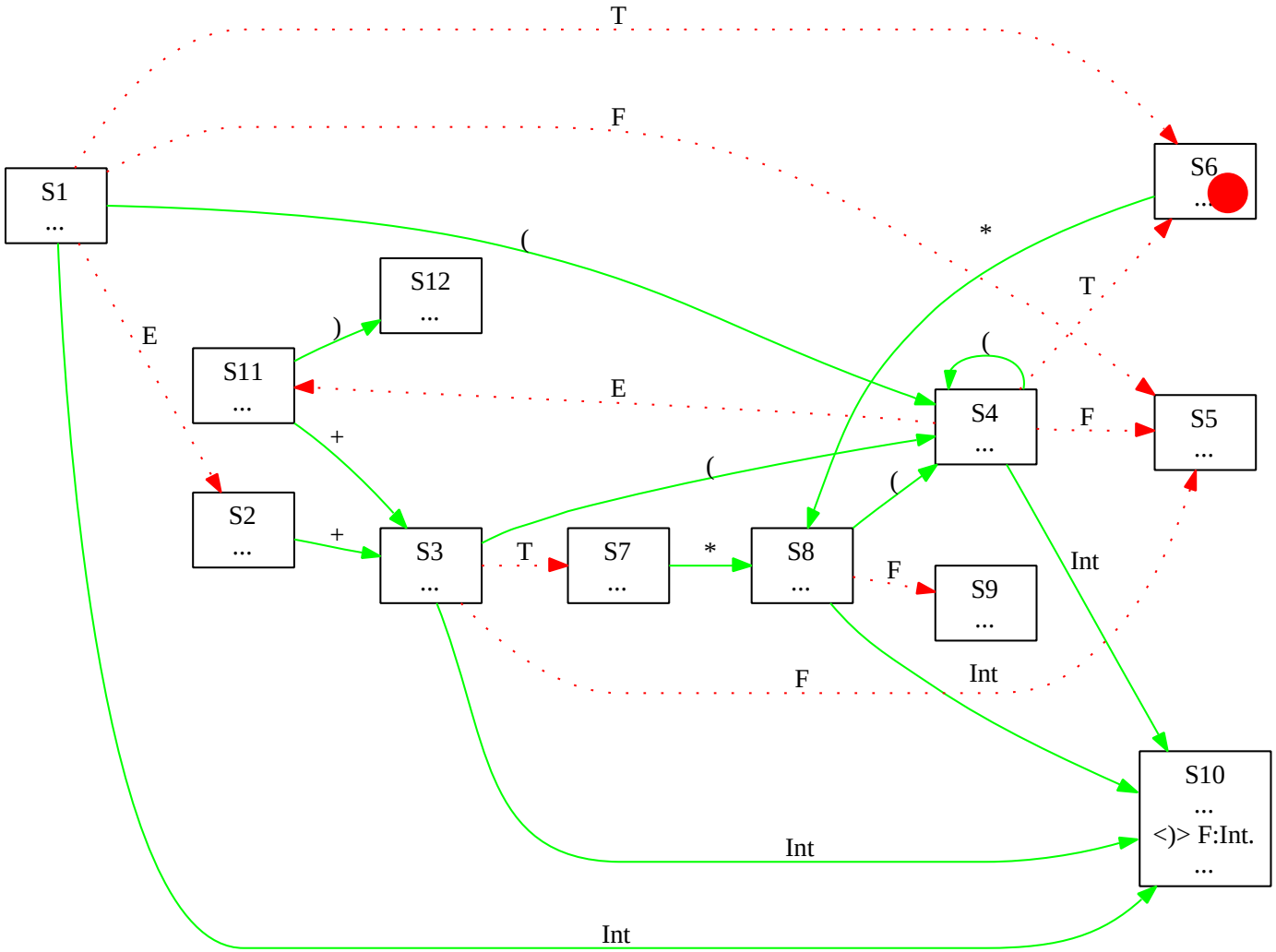
Entrée	Pile	État	Prochaine action
) $	S_1 (S_4 **int**	S_{10}	reduce **int** $\longrightarrow$ F, goto F



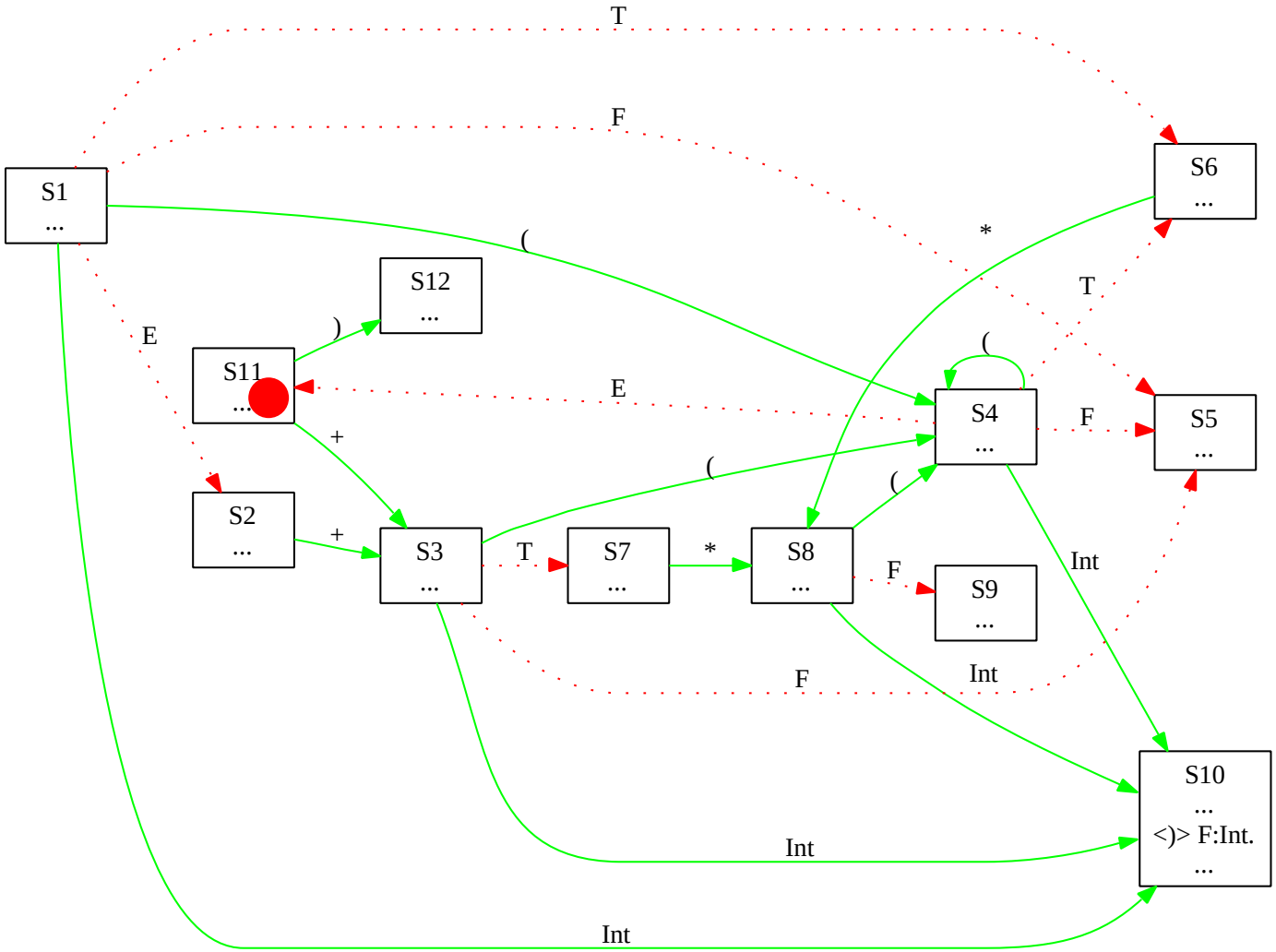
Entrée	Pile	État	Prochaine action
) $	S_1 (S_4 F		goto F



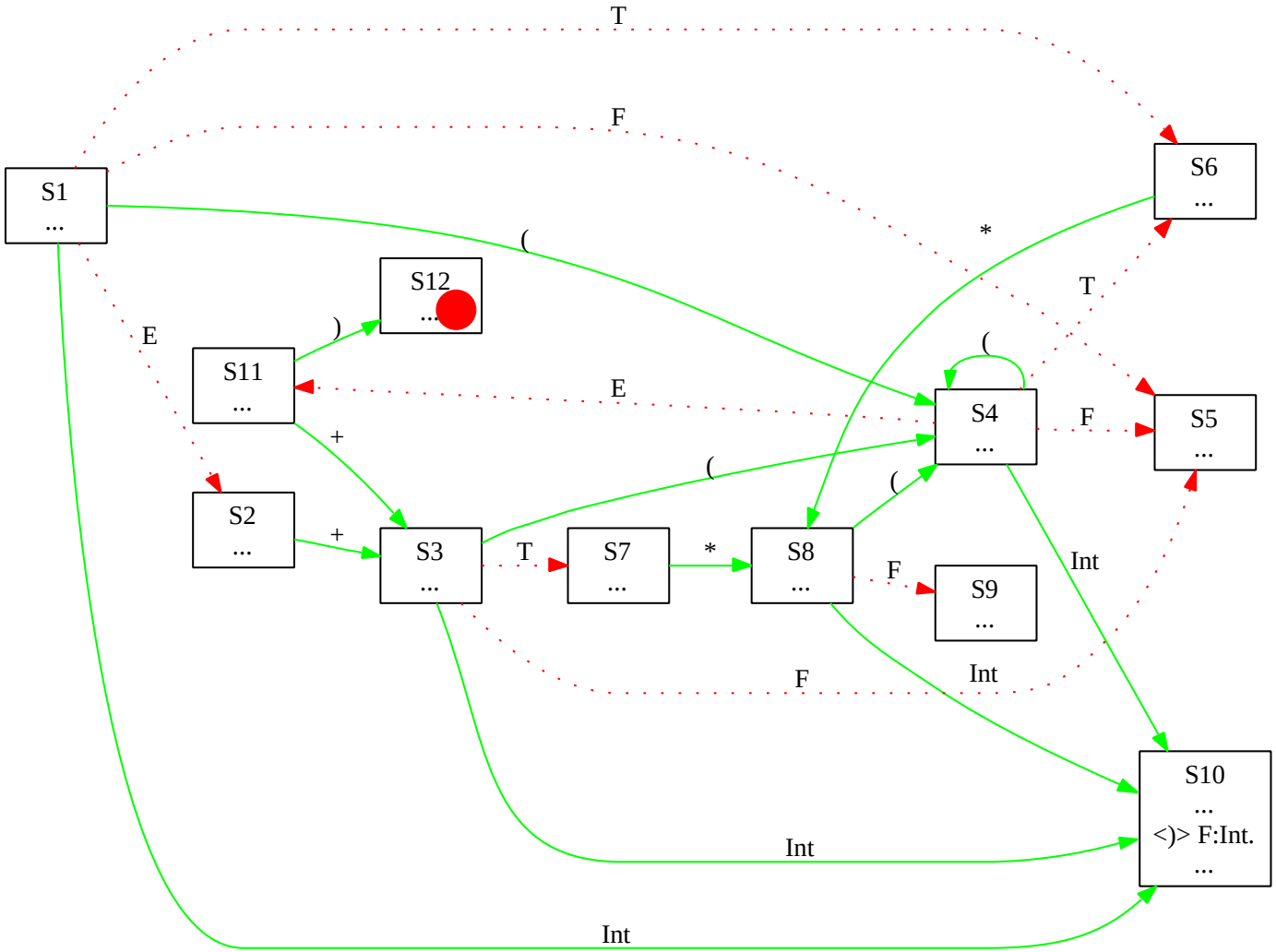
Entrée	Pile	État	Prochaine action
) $	S_1 (S_4 F	S_5	reduce $F \longrightarrow T$, goto T



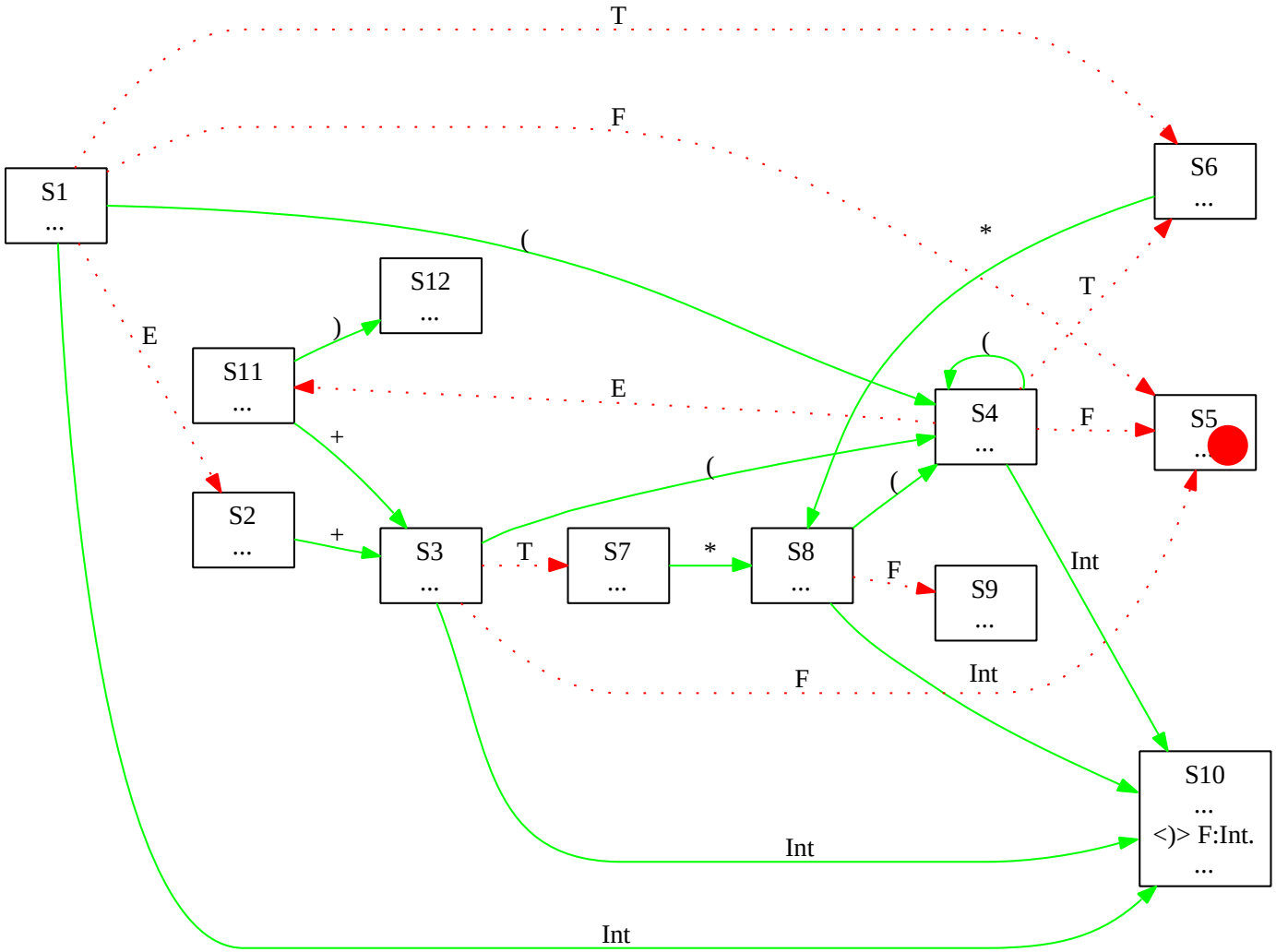
Entrée	Pile	État	Prochaine action
) $	S_1 (S_4 T	S_6	reduce $T \longrightarrow E$, goto E



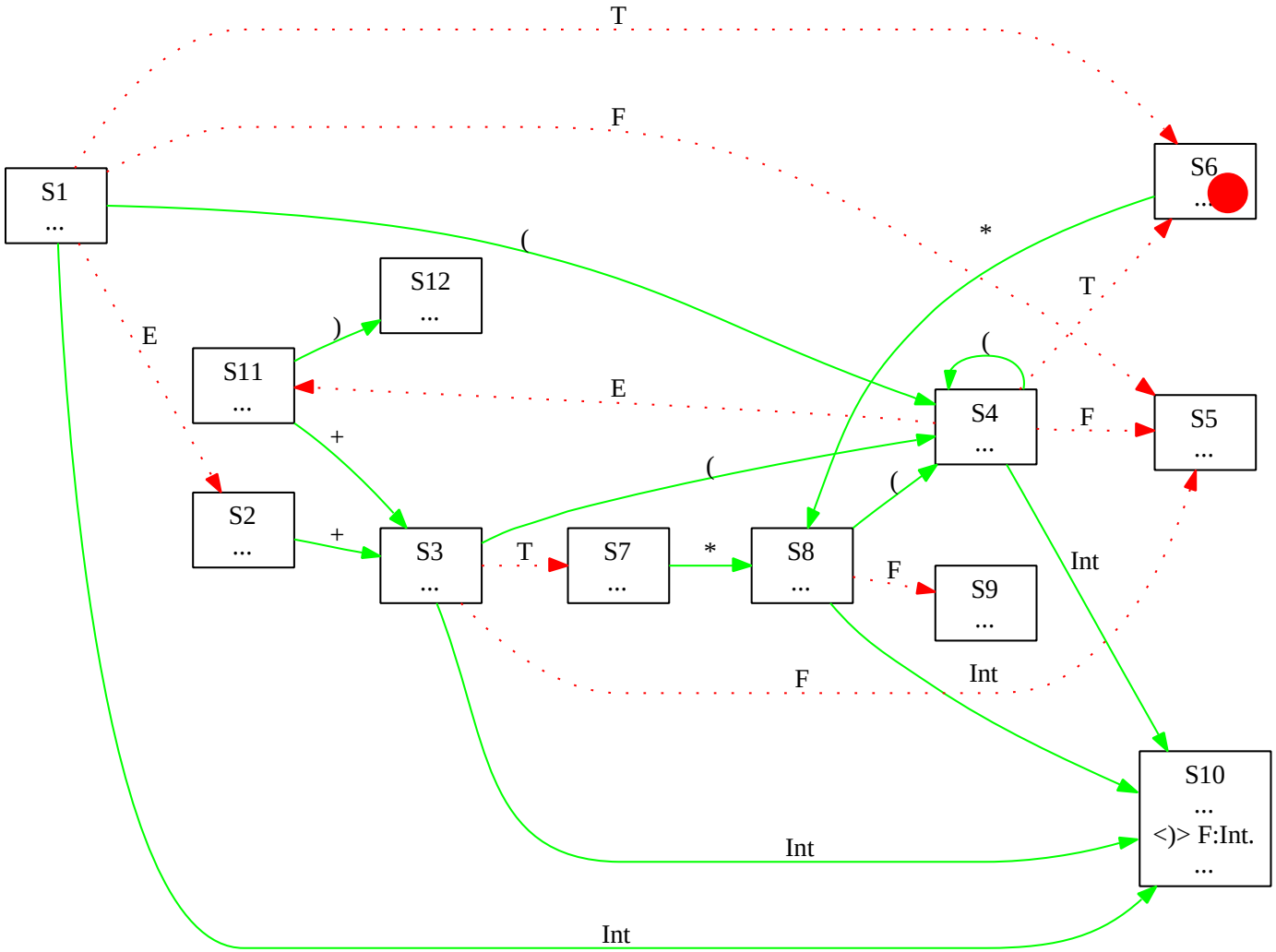
Entrée	Pile	État	Prochaine action
) $	S_1 (S_4 E	S_{11}	shift S_{12}



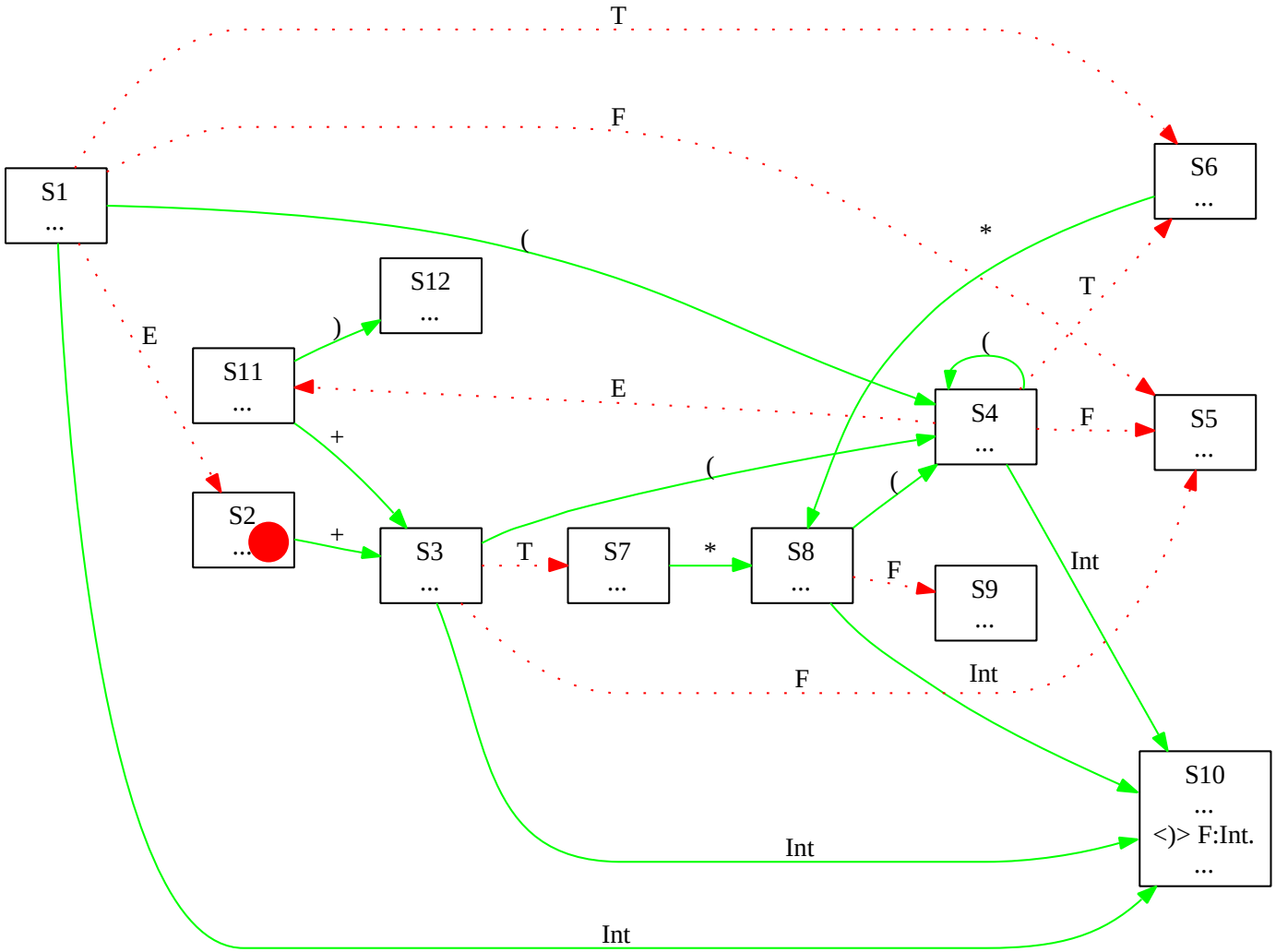
Entrée	Pile	État	Prochaine action
$	S_1 (S_4 E S_{11})	S_{12}	reduce $(E) \longrightarrow F$, goto F



Entrée	Pile	État	Prochaine action
$	S_1 F	S_5	reduce $F \longrightarrow T$, goto T



Entrée	Pile	État	Prochaine action
$	S_1 T	S_6	reduce $T \longrightarrow E$, goto E



Entrée	Pile	État	Prochaine action
$	S_1 *E*	S_2	*accept*

Que produisent les outils existants ?

Yacc, Bison, etc. produisent des programmes C, donc *sans garantie de sûreté.* Ils utilisent une *union* pour représenter les valeurs sémantiques.

ML-Yacc ou Happy produisent des programmes ML ou Haskell, donc typés. Néanmoins, *le typage n'empêche pas la levée d'une exception* pendant l'exécution, en cas d'échec d'un filtrage.

À quoi ressemble le code produit par ces derniers ?

Interface avec l'analyseur lexical

Les *lexèmes* sont constitués d'une étiquette et éventuellement d'une valeur sémantique:

type *token* = *KPlus* | *KStar* | *KLeft* | *KRight* | *KEnd* | *KInt* of *int*

L'analyseur lexical fournit deux fonctions pour consulter et jeter le lexème courant:

val *peek* : *unit* → *token*
val *discard* : *unit* → *unit*

Structures de données

Le type des états se définit aisément:

type *state* = S0 | S1 | ... | S11

Structures de données (suite)

La pile est formée de paires d'un état et d'une valeur sémantique dont le type dépend du non-terminal auquel elle est associée. Il s'agit donc d'une liste chaînée de cellules *étiquetées*.

```
type stack =
  | SEmpty
  | SPlus of stack × state
  | SStar of stack × state
  | SLeft of stack × state
  | SRight of stack × state
  | SEnd of stack × state
  | SInt of stack × state × int
  | SE of stack × state × int
  | ST of stack × state × int
  | SF of stack × state × int
```

Implantation (structure générale)

L'automate est implanté par une fonction *run* qui, étant donnés l'état, la pile et (implicitement) le flot de lexèmes courants, produit la valeur sémantique correspondant au mot complet ou bien échoue.

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ... (* transitions shift ou reduce *)
  | _, _ →
      raise SyntaxError
```

Implantation (décalage)

Une transition *shift* empile état courant et valeur sémantique associée au lexème courant, consomme ce dernier, et modifie l'état courant:

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ...
  | S9, KStar → (* shift S7 *)
      discard ();
      run S7 (SStar (stack, S9))
  | ...
```

Implantation (réduction)

Une transition *reduce* dépile un certain nombre de valeurs sémantiques, les exploite pour en calculer une nouvelle, et empile celle-ci.

```
let rec run (s : state) (stack : stack) : int =
  match s, peek() with
  | ...
  | S9, KPlus → (* reduce E{x} + T{y} → E{x + y} *)
      let ST (SPlus (SE (stack, s, x), _), _, y) = stack in
      let stack = SE (stack, s, x + y) in
      gotoE s stack (* goto E *)
  | ...
```

On observe ici un *filtrage non exhaustif*.

Implantation (fin)

Une transition *goto* se base sur l'état extrait de la pile pendant la réduction pour déterminer le prochain état courant.

```
and gotoE (s : state) : stack → int =
  match s with
  | S0 →
      run S1
  | S4 →
      run S8
```

On observe un nouveau *filtrage non exhaustif*.

En résumé

Ce programme est considéré par le compilateur ML comme bien typé. Néanmoins, celui-ci affiche des avertissements pour filtrage non exhaustif, ce qui signifie que *l'absence d'échec à l'exécution n'est pas garantie.*

Le problème est de réécrire le programme de façon à n'effectuer que des filtrages exhaustifs. En supprimant les tests dynamiques redondants, on obtiendra une *garantie de sûreté* et une meilleure *efficacité*.

Pourquoi ces tests sont-ils redondants ?

Les tests dynamiques effectués lors de la transition *reduce* précédente sont redondants parce que, lorsque l'automate est dans l'état S_9, la pile est nécessairement de la forme

$$\ldots \; ? \; E \; ? \; + \; ? \; T$$

Les tests dynamiques effectués lors de la transition *goto E* précédente sont redondants parce que, lorsque l'automate est dans l'état S_9, la pile est nécessairement de la forme

$$\ldots \; (S_0 \mid S_4) \; ? \; ? \; ? \; ? \; ?$$

L'invariant (fragment)

En fait, on peut démontrer que, lorsque l'automate est dans l'état S_9, la pile est nécessairement de la forme

$$\ldots \quad (S_0 \mid S_4) \quad E \quad (S_1 \mid S_8) \quad + \quad S_6 \quad T$$

Plus généralement, la connaissance de l'état courant détermine celle d'un *suffixe* de la pile...

L'invariant complet

Pile							État
ϵ							S_0
ϵ	S_0	E					S_1
...	$(S_0 \mid S_4)$	T					S_2
...	$(S_0 \mid S_4 \mid S_6)$	F					S_3
...	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	(					S_4
...	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	**int**					S_5
...	$(S_0 \mid S_4)$	E	$(S_1 \mid S_8)$	+			S_6
...	$(S_0 \mid S_4 \mid S_6)$	T	$(S_2 \mid S_9)$	*			S_7
...	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	(	S_4	E			S_8
...	$(S_0 \mid S_4)$	E	$(S_1 \mid S_8)$	+	S_6	T	S_9
...	$(S_0 \mid S_4 \mid S_6)$	T	$(S_2 \mid S_9)$	*	S_7	F	S_{10}
...	$(S_0 \mid S_4 \mid S_6 \mid S_7)$	(	S_4	E	S_8	)	S_{11}

Vers un typage plus fin

On démontre *manuellement* sans difficulté, par induction structurelle sur une exécution de l'automate, la correction de l'invariant.

Pour que le compilateur puisse exploiter cet invariant, il faut qu'on le lui fournisse *explicitement*, et qu'il le vérifie *mécaniquement*.

Le langage de programmation doit être doté d'un système de types suffisamment expressif pour permettre le codage de l'invariant.

L'idée

Il faut expliquer au compilateur qu'il y a *corrélation* entre état courant et structure de la pile.

Pour cela, nous paramétrons le type *state* par une variable de types *a*. L'idée est que *si l'état courant admet le type a state, alors la pile courante est de type a*.

Structure des piles

Le type *stack* disparaît. La structure des piles sera définie à l'aide d'une famille de types paramétrés définis de façon mutuellement indépendante:

```
type empty = SEmpty
type a cellPlus = SPlus of a × a state
type a cellStar = SStar of a × a state
type a cellLeft = SLeft of a × a state
type a cellRight = SRight of a × a state
type a cellInt = SInt of a × a state
type a cellE = SE of a × a state × int
type a cellT = ST of a × a state × int
type a cellF = SF of a × a state × int
```

(Comparer à la définition originale.)

Codage de l'invariant (fragment)

Le fait, lorsque l'automate est dans l'état S_9, la pile est nécessairement de la forme

$$\ldots \ ? \ E \ ? \ + \ ? \ T,$$

sera codé en attribuant à la constante S9 le type

$$\forall a.a \ \mathit{cellE} \ \mathit{cellPlus} \ \mathit{cellT} \ \mathit{state}$$

et de même pour les autres états.

Une telle déclaration est impossible en ML! Le type *state* sera un *type de données algébrique généralisé* ou *gardé* (GADT).

Structure des états

```
type state : * ⟶ * where
| S0 : empty state
| S1 : empty cellE state
| S2 : ∀a.a cellT state
| S3 : ∀a.a cellF state
| S4 : ∀a.a cellLeft state
| S5 : ∀a.a cellInt state
| S6 : ∀a.a cellE cellPlus state
| S7 : ∀a.a cellT cellStar state
| S8 : ∀a.a cellLeft cellE state
| S9 : ∀a.a cellE cellPlus cellT state
| S10 : ∀a.a cellT cellStar cellF state
| S11 : ∀a.a cellLeft cellE cellRight state
```

Implantation (structure générale)

Le type de la fonction *run* est modifié: elle accepte un état arbitraire et une pile *dont la structure est cohérente vis-à-vis de cet état.*

```
let rec run : ∀a.a state → a → int =
  fun s stack →
    match s, peek() with
    | ...
    | _, _ →
        raise SyntaxError
```

(Comparer au type original.)

Implantation (décalage)

Le code d'une transition *shift* est inchangé, mais la vérification de types devient plus subtile.

```
let rec run : ∀a.a state → a → int =
  fun s stack →
    match s, peek() with
    | S9, KStar →
        (* SStar (stack, S9) a le type a cellStar *)
        (* run S7 a le type ∀γ.γ cellT cellStar → int *)
        (* Or a = β cellE cellPlus cellT, pour β inconnu *)
        (* Donc a cellStar = γ cellT cellStar, pour γ = β cellE cellPlus *)
        discard ();
        run S7 (SStar (stack, S9))
```

(Consulter la définition du type des états.)

Implantation (réduction)

Le code d'une transition *reduce* est également inchangé, mais *le filtrage est maintenant exhaustif*.

```
let rec run : ∀a.a state → a → int =
  fun s stack →
    match s, peek() with
    | S9, KPlus →
        (* a = β cellE cellPlus cellT, pour β inconnu *)
        (* Donc stack : β cellE cellPlus cellT *)
        let ST (SPlus (SE (stack, s, x), _), _, y) = stack in
        (* stack : β, s : β state, x : int, y : int *)
        let stack = SE (stack, s, x + y) in
        (* stack : β cellE *)
        gotoE s stack
```

Implantation (fin)

Le type attribué à *gotoE* indique qu'au sommet de pile doit se trouver une cellule associée au non-terminal E et que le reste de la pile doit être cohérent avec l'état s.

```
and gotoE : ∀a.a state → a cellE → int =
  fun s →
    match s with
    | S0 →
        run S1
    | S4 →
        (* run S8 a le type β cellLeft cellE → int, pour tout β *)
        (* Or a = β cellLeft, pour β inconnu *)
        run S8
```

(Ce filtrage reste non exhaustif.)

En résumé

Nous avons codé une partie de l'invariant dans les déclarations de types de données et dans les types attribués aux fonctions *run* et *goto*. On pourrait coder l'intégralité de l'invariant.

Alors, la vérification des types *contient la démonstration* de l'invariant.

Le filtrage fournit localement *de nouvelles hypothèses*, que l'on peut exploiter pour établir une égalité entre types.

Résultats

Nous avons obtenu une *garantie de sûreté* à propos de l'analyseur syntaxique engendré, sans devoir exiger une *confiance* en la correction du générateur.

L'outil qui produit l'automate *connaît l'invariant, ou croit le connaître,* et engendre sans difficulté les déclarations de types appropriées.

Si l'outil produit un programme incorrect, celui-ci sera rejeté par le compilateur.

Il reste nécessaire d'avoir confiance en la correction du compilateur, à moins d'employer un compilateur certifiant.

Mieux prouver nos programmes

Nous avons exploité un système de types très expressif pour prouver la *sûreté* d'un programme.

Des *assistants à la preuve*, comme Coq, permettent cela depuis longtemps. Ici, cependant, nous sommes restés dans le cadre d'un *langage de programmation* doté, en particulier, d'un mécanisme d'inférence de types puissant et d'un schéma de compilation très efficace.

Réduire le fossé entre programmation et preuve reste l'objet de recherches actives.

Références

Transparents de cet exposé, version préliminaire de l'article, prototype du vérificateur de types et prototype du générateur d'analyseurs syntaxiques sont disponibles en ligne:

http://cristal.inria.fr/~fpottier/
http://cristal.inria.fr/~regisgia/