Chapitre 3 Théorie des syntaxes abstraites

Idées

- syntaxe abstraite = ensemble d'arbres (AST)
- sémantique/interprétation d'un AST = parcours récursif de l'AST
- Ici, arbres codés comme des mots particuliers (comme dans n'importe quel langage de prog).
  NB : point de vue TL et pas théorie des graphes...
  codage des arbres par notation préfixe (le +simple).

Pour commencer : des exemples !

Signature d'un ensemble d'AST pour expr. rég. sur $\{a, b\}$

Signature = liste de *constructeurs* (opérateurs) avec leur type.

nom du type des expr. rég. sur $\{a, b\}$ $\qquad \mathtt{Rexp}$

constants pour $c \in \{\mathtt{void}, \mathtt{epsilon}, a, b\}$,

$$c : \mathtt{Rexp}$$

unaire $\qquad \mathtt{star} : \mathtt{Rexp} \to \mathtt{Rexp}$

binaires pour $c \in \{\mathtt{u}, \mathtt{dot}\}$,

$$c : \mathtt{Rexp} \times \mathtt{Rexp} \to \mathtt{Rexp}$$

Exo 3.1† Dessiner l'AST de l'expr. rég. "$b + (a.a.(\epsilon + a)^*)$".
Coder cet AST en notation préfixe (suite d'appels des constructeurs sans parenthèse).

Exo 3.2† Dessiner l'AST encodé par le mot "$\mathtt{star}\ \mathtt{u}\ \mathtt{dot}\ a\ b\ a$".

Exo 3.3† Donner une BNF qui définit cette notation préfixe d'AST.

Récursion structurelle sur une structure d'AST

Idée pour "*définir simplement*" une fonction de $\texttt{Rexp} \rightarrow D$, choisir

- constantes A_c dans D, pour $c \in \{\texttt{void}, \texttt{epsilon}, a, b\}$.
- une fonction $A_{\texttt{star}} \in D \rightarrow D$.
- deux fonctions $A_{\texttt{u}}$ et $A_{\texttt{dot}}$ de $D \times D \rightarrow D$,

Exo 3.4† Appliquer cette idée avec

- $D \stackrel{def}{=} \mathcal{P}(V^*)$ pour la sémantique des Rexp (comme ensemble de mots).
- $D \stackrel{def}{=}$ ensemble des automates finis (non-déterministes avec ϵ-transitions) pour générer l'automate associé à une Rexp. Calculer l'automate de $A_{\texttt{dot}}(A_a, A_a)$ avec vos définitions.

Récursion sur AST par système d'attributs

Exemple de système d'attributs (sur `Rexp`)
Principe : BNF décoré avec *attributs*.
Exprime calculs sur l'arbre.

Typage de l'attribut : $\texttt{Rexp}{\uparrow}\mathcal{P}(\{a, b\}^*)$.

$$
\begin{array}{lcll}
\texttt{Rexp}{\uparrow}\ell & ::= & \texttt{void} & \ell := \emptyset \\
 & | & \texttt{epsilon} & \ell := \{\epsilon\} \\
 & | & a & \ell := \{a\} \\
 & | & b & \ell := \{b\} \\
 & | & \texttt{u}\ \texttt{Rexp}{\uparrow}\ell_1\ \texttt{Rexp}{\uparrow}\ell_2 & \ell := \ell_1 \cup \ell_2 \\
 & | & \texttt{dot}\ \texttt{Rexp}{\uparrow}\ell_1\ \texttt{Rexp}{\uparrow}\ell_2 & \ell := \ell_1.\ell_2 \\
 & | & \texttt{star}\ \texttt{Rexp}{\uparrow}\ell_1 & \ell := \ell_1^*
\end{array}
$$

Exo 3.5[†] Dessiner la propagation de l'attribut sur l'AST encodé par le mot "`star u dot` *a b a*".

Deux modes de passage des attributs

- attribut $\uparrow$ dit "synthétisé" (propagé du fils vers le père)
  $\Rightarrow$ correspond à un "résultat" de l'appel récursif.
- attribut $\downarrow$ dit "hérité" (propagé du père vers le fils)
  $\Rightarrow$ correspond à un paramètre d'entrée de l'appel récursif.

Exo 3.6† Soient les arbres binaires de sorte `B` engendrés par constructeurs "$\mathtt{l} : \mathtt{B}$" et "$\mathtt{n} : \mathtt{B} \times \mathtt{B} \to \mathtt{B}$".
Dans chacun des cas ci-dessous, définir système d'attributs qui :

1. compte le nombre de feuilles (noeuds `l`) dans l'arbre.
2. indique si toutes les feuilles de l'arbre sont à une profondeur h ou $h-1$, avec h paramètre donné (par convention, feuille de hauteur 0).

Appliquer les algorithmes sous-jacents sur les exemples ci-dessous en dessinant la propagation d'attributs sur chaque nœuds :

1. `n l n n l l n l l`
2. `n n l l n n l l n l l`

pour Q2, on prendra h valant à la racine 2, puis ensuite 3.

Exercices

Exo 3.7† On considère le système donné à la tâche 4 du TP.

1. Soit p l'AST "*and var* 1 *neg var* 2".
   Calculer le r retourné une dérivation de "$p{\downarrow}1{\uparrow}r$".
2. Calculer la NNF attendue pour "`-(((t&1)>(-2|f))&3)`".

Exo 3.8† Exo 3 de mars 2018.

Applications des systèmes d'attributs à la programmation

- Interpréteur sur des arbres : cf. tâche 4 du TP.

- Interpréteur sur des mots :
  programmation d'un analyseur en notation préfixe.
  $\Rightarrow$ la suite de ce cours (et aussi tâche 5 du TP).

Chapitre 3 Théorie des syntaxes abstraites

Plan de la section 3.2

Terminologie de cette définition mathématique

- Définition inspirée de notion de "*termes*" en logique :

  1 arbre de syntaxe
  = 1 terme
  = 1 mot dans alphabet des *constructeurs* d'arbres
  = 1 mot en **notation préfixe** (Łukasiewicz 1924)

  cf. école polonaise de logique des années 1920-1930 (Tarski).

- "Forme" des termes définie par *signature multisortée*.
  1 sorte = 1 nom de type d'arbre = 1 catégorie syntaxique.
  On peut avoir +sieurs sortes.

- Syntaxe abstraite = *langage de termes* engendré par une *signature multisortée* fixée.

NB : notation préfixe aussi une syntaxe concrète qui peut suffire pour fichiers binaires (non lus par humain).

Définition des signatures multisortées

$\Sigma \overset{def}{=} (\mathcal{C}_{ext}, \mathcal{C}_{int}, \mathcal{S}_{ext}, \mathcal{S}_{int}, s_{princ}, \mathrm{type})$ avec

- $\mathcal{C}_{int}$: ensemble *fini* de symboles de "*constructeurs internes*".
- $\mathcal{C}_{ext}$: ens. *dénombrable* de "*constructeurs externes*".
- $\mathcal{C} \overset{def}{=} \mathcal{C}_{ext} \uplus \mathcal{C}_{int}$ est l'alphabet du langage
  (termes $\subseteq$ mots de $\mathcal{C}^+$).
- $\mathcal{S} \overset{def}{=} \mathcal{S}_{ext} \uplus \mathcal{S}_{int}$ ensemble *fini* de symboles de "*sortes*".
  1 sorte $=$ nom d'un type d'arbre.
- s_{princ} de $\mathcal{S}_{int}$ $=$ "*la sorte principale*".
- type fonction totale de $\mathcal{C} \to \mathcal{S}^+$
  NB "$\mathrm{type}(c) = s_1.\cdots.s_{n+1}$" formalise la notation :
  $$c : s_1 \times \ldots \times s_n \to s_{n+1}$$
- **Conditions supplémentaires**
  pour tout c de $\mathcal{C}_{ext}$, $\mathrm{type}(c) \in \mathcal{S}_{ext}$
  pour tout c de $\mathcal{C}_{int}$, $\mathrm{image}(\mathrm{type}(c)) \in \mathcal{S}_{int}$

Exemple de signature Σ_{TP} pour **Prop** (similaire au TP)

- $\mathcal{S}_{ext} \stackrel{def}{=} \{\texttt{Pos}\}$ et $\mathcal{S}_{int} \stackrel{def}{=} \{\texttt{Prop}\}$.
- $s_{princ} \stackrel{def}{=} \texttt{Prop}$.
- $\mathcal{C}_{ext} \stackrel{def}{=} \mathbb{N}\backslash\{0\}$ et $\mathcal{C}_{int} \stackrel{def}{=} \{\texttt{true}, \texttt{false}, \texttt{var}, \texttt{neg}, \texttt{and}, \texttt{or}\}$.
- type défini par :
  - pour $c \in \mathbb{N}\backslash\{0\}$, $c : \texttt{Pos}$ (constructeurs externes).
  - pour $c \in \{\texttt{true}, \texttt{false}\}$, $c : \texttt{Prop}$.
  - $\texttt{var} : \texttt{Pos} \rightarrow \texttt{Prop}$.
  - $\texttt{neg} : \texttt{Prop} \rightarrow \texttt{Prop}$.
  - pour $c \in \{\texttt{and}, \texttt{or}, \texttt{implies}\}$, $c : \texttt{Prop} \times \texttt{Prop} \rightarrow \texttt{Prop}$.

Exo 3.9† Donner la BNF associée à cette signature Σ_{TP}.

Exo 3.10† Définir une signature pour NNF avec sortes **Nnf** et **Ncst**, puis la BNF associée

Définition des termes sur signature Σ

Système d'équation algébrique associée à Σ

- une variable X_s pour chaque sorte $s \in S$
- pour chaque sorte s une équation

$$X_s = \bigcup_{c:s_1 \times \ldots \times s_n \to s} \{c\}.X_{s_1}.\cdots.X_{s_n}$$

Langage des termes de sorte s
On note $(T_\Sigma(s))_{s \in S}$ le plus petit point-fixe de ce système d'équations.

Syntaxe abstraite = $T_\Sigma(s_{princ})$ (abbrev. T_Σ)

Fonctions constructeurs (au sens des AST)

Étant donné Σ une signature multisortée et $c \in \mathcal{C}$ tel que $c : s_1 \times \ldots \times s_n \rightarrow s$

Définition La *fonction constructeur* de c, notée $\bar{c}$, est la *fonction totale* de $(\mathcal{C}^+)^n \rightarrow \mathcal{C}^+$ définie par

$$\bar{c}(t_1, \ldots, t_n) \stackrel{def}{=} c.t_1.\cdots.t_n$$

Exo 3.11 Montrer $n \geq 2 \Rightarrow \bar{c}$ pas injective sur $\overbrace{\mathcal{C}^+ \times \ldots \times \mathcal{C}^+}^{n} \rightarrow \mathcal{C}^+$

Exo 3.12 Montrer les deux lemmes suivants.

Lemme du préfixe unique Pour tout mot u de $\mathcal{C}^*$, il existe *au plus un* préfixe t de u qui vérifie il existe $s \in S$ tq $t \in T_\Sigma(s)$.

Corollaire (injectivité des constructeurs)
$\bar{c}$ est *injective* sur $T_\Sigma(s_1) \times \ldots \times T_\Sigma(s_n) \rightarrow T_\Sigma(s)$.
NB : justifie *unicité* du "parenthésage implicite" !

Plan de la section 3.2

Motivation

Systèmes d'attributs :	méta-syntaxe pour définir *simultanément* syntaxe abstraite + sémantique
Leur méta-sémantique :	1 signature Σ et sa Σ-sémantique (thm de récursion structurelle)

- Systèmes d'attributs inventés par Knuth en 1967 (en fait plus généraux que ceux présentés ici).
  une notation pratique pour *concevoir* un nouveau langage : on conçoit l'AST et la sémantique simultanément, *avant* la syntaxe concrète.
- Σ-sémantique = une formalisation mathématique simple de la notion de sémantique (ou d'interpréteur) d'une structure d'AST.

Définition des Σ-sémantiques (ou Σ-algèbres, Σ-structures)

Définition Σ-sémantique = un couple $((D_s)_{s\in\mathcal{S}}, (A_c)_{c\in\mathcal{C}})$ où :

- $(D_s)_{s\in\mathcal{S}}$ est une suite d'ensembles ("domaines sémantiques").
- pour tout $c \in \mathcal{C}$, si $c : s_1 \times \ldots \times s_n \to s$,
  alors A_c fonction totale de $D_{s_1} \times \ldots \times D_{s_n} \to D_s$.
  (A_c "action sémantique").

Théorème de la récursion structurelle
Étant donné une Σ-sémantique, il existe une unique

$$[\![.]\!] \text{ fonction totale de } (\bigcup_{s\in\mathcal{S}} T_\Sigma(s)) \to (\bigcup_{s\in\mathcal{S}} D_s)$$

telle que :

- pour tous $s \in \mathcal{S}$ et $t \in T_\Sigma(s)$, on a $[\![t]\!] \in D_s$
- $[\![c.t_1.\cdots.t_n]\!] = A_c([\![t_1]\!], \ldots, [\![t_n]\!])$.

Rem on peut voir $[\![.]\!]$ comme ensemble fini de fonctions *mutuellement récursives* notées $[\![.]\!]_s \in T_\Sigma(s) \to D_s$ pr tt s de S.

(Méta)syntaxe et sémantique des BNF attribuées d'AST

BNFs attribuées d'AST :
1 (méta)notation pour définir signature Σ et sa Σ-sémantique

Principe

- non-terminal = sorte
- Chaque profil de non-terminal "$s{\downarrow}E_1 \ldots {\downarrow}E_m{\uparrow}E'_1 \ldots {\uparrow}E'_{m'}$" définit $D_s \overset{def}{=} E_1 \times \ldots \times E_m \rightarrow E'_1 \times \ldots \times E'_{m'}$.
- L'équation (avec attributs) du non-terminal s définit à la fois le typage des constructeurs d'image s et l'action sémantique associée à ces constructeurs.

Les 4 diapos suivantes définissent :

- la (méta)syntaxe des équations
- la (méta)sémantique des équations avec attributs

Les “alternatives” d’une équation vues comme des “règles”

Pour simplifier formalisation, abstraction de “::=” et “|” en “→”.

Exemple

$$
\begin{array}{lcll}
\texttt{Rexp}\uparrow\ell & \to & \texttt{void} & \ell := \emptyset \\
\texttt{Rexp}\uparrow\ell & \to & \texttt{epsilon} & \ell := \{\epsilon\} \\
\texttt{Rexp}\uparrow\ell & \to & a & \ell := \{a\} \\
\texttt{Rexp}\uparrow\ell & \to & b & \ell := \{b\} \\
\texttt{Rexp}\uparrow\ell & \to & \texttt{u Rexp}\uparrow\ell_1\ \texttt{Rexp}\uparrow\ell_2 & \ell := \ell_1 \cup \ell_2 \\
\texttt{Rexp}\uparrow\ell & \to & \texttt{dot Rexp}\uparrow\ell_1\ \texttt{Rexp}\uparrow\ell_2 & \ell := \ell_1 . \ell_2 \\
\texttt{Rexp}\uparrow\ell & \to & \texttt{star Rexp}\uparrow\ell_1 & \ell := \ell_1^*
\end{array}
$$

Ainsi, chaque alternative d’une équation correspond à une *règle*

$$s\,p\,p' \to c \quad s_1 p_1 p_1' \quad \ldots \quad s_n p_n p_n' \quad \mathcal{A}$$

où p et $(p_i)_{i\in[1,n]}$ sont des listes d’attributs hérités,
où p' et $(p_i')_{i\in[1,n]}$ sont des listes d’attributs synthétisés,
et où $\mathcal{A}$ est une *action* qui effectue une séquence de *définitions* (notées comme des affectations).

Formalisation de la notion de règle

Une règle

$$s_{n+1}\, p_{n+1}\, p'_{n+1} \rightarrow c \quad s_1 p_1 p'_1 \quad \ldots \quad s_n p_n p'_n \quad \mathcal{A}$$

- **définit** une règle de typage $c : s_1 \times \ldots \times s_n \rightarrow s_{n+1}$.
- **doit vérifier** pour tout $i \in [1, n+1]$,
  si "$s_i {\downarrow} E_1 \ldots {\downarrow} E_m {\uparrow} E'_1 \ldots {\uparrow} E'_{m'}$"
  alors
  - *p_i est liste de noms* "${\downarrow} x_1 \ldots {\downarrow} x_m$"
    qui représente un élément $(x_1, \ldots, x_m)$ de $E_1 \times \ldots \times E_m$;
  - de même pour p'_i avec ${\uparrow} x'_1 \ldots {\uparrow} x'_{m'}$.
- **doit aussi vérifier** des conditions sur $\mathcal{A}$ (voir diapo suivante)
- **définit alors** une fonction $A_c \in D_{s_1} \times \ldots \times D_{s_n} \rightarrow D_{s_{n+1}}$
  spécifiée par $\mathcal{A}$ (voir diapo suivante)

Action associée à une règle

Une règle "$s\, p\, p' \to c \quad s_1 p_1 p'_1 \quad \ldots \quad s_n p_n p'_n \quad \mathcal{A}$" définit une fonction $A_c \in D_{s_1} \times \ldots \times D_{s_n} \to D_s$" telle que $A_c(f_1, \ldots, f_n)(p)$ retourne le p' calculé par $\mathcal{A}$ **sous la condition** que :

- $\mathcal{A}$ définisse les noms de $(p_i)_{i \in [1,n]}$ *puis* de p' en fonction de ceux p et $(p'_i)_{i \in [1,n]}$,
- et que pour $i \leq n$, tous les noms de p_i soient définis *avant* toute utilisation d'un nom de p'_i.

Car, implicitement, pour $i \leq n$, les noms de p'_i sont définis à la *première* utilisation d'un de ses noms par "$(x'_1, \ldots, x'_{m'}) := f_i(x_1, \ldots, x_m)$".

NB : cet appel à f_i représente un *appel récursif* sur le i-ème fils du constructeur c.

Exemple d'action associée à une règle

Pour règle

$$s{\downarrow}x{\uparrow}x' \to c \quad s_1{\downarrow}x_1{\uparrow}x'_1 \quad s_2{\downarrow}x_2{\uparrow}x'_2 \qquad \begin{array}{l} x_1 := g_1(x) \ ; \\ x_2 := g_2(x'_1, x) \ ; \\ x' := g(x_1, x'_1, x'_2) \end{array}$$

on obtient $A_c(f_1, f_2) = x \mapsto \begin{array}{l} \texttt{soit } x_1 \stackrel{def}{=} g_1(x) \ , \\ \texttt{soit } x'_1 \stackrel{def}{=} f_1(x_1) \ , \\ \texttt{soit } x_2 \stackrel{def}{=} g_2(x'_1, x) \ , \\ \texttt{soit } x'_2 \stackrel{def}{=} f_2(x_2) \ , \\ \texttt{retourner } g(x_1, x'_1, x'_2) \end{array}$

Même règle avec du "sucre" (noms remplacés par leur def) :

$$s{\downarrow}x{\uparrow}g(g_1(x), x'_1, x'_2) \to c \quad s_1{\downarrow}g_1(x){\uparrow}x'_1 \quad s_2{\downarrow}g_2(x'_1, x){\uparrow}x'_2$$

Chapitre 3 Théorie des syntaxes abstraites

Exemple d'une autre signature Σ_{AST} pour **Prop**

- $\mathcal{S}_{ext} \stackrel{def}{=} \{\mathtt{Bool}, \mathtt{Op}, \mathtt{Pos}\}$.
- $\mathcal{S}_{int} \stackrel{def}{=} \{\mathtt{Prop}\}$ et $s_{princ} \stackrel{def}{=} \mathtt{Prop}$.
- $\mathcal{C}_{ext} \stackrel{def}{=} \{\mathtt{true}, \mathtt{false}, \mathtt{and}, \mathtt{or}\} \uplus \mathbb{N}\backslash\{0\}$ avec $\mathbb{N}\backslash\{0\} \stackrel{def}{=} \mathbb{N}\backslash\{0\}$.
- $\mathcal{C}_{int} \stackrel{def}{=} \{\mathtt{neg}, \mathtt{var}, \mathtt{cte}, \mathtt{op}\}$
- type défini par :
  - pour $c \in \{\mathtt{true}, \mathtt{false}\}$, $c : \mathtt{Bool}$.
  - pour $c \in \mathbb{N}\backslash\{0\}$, $c : \mathtt{Pos}$.
  - pour $c \in \{\mathtt{and}, \mathtt{or}, \mathtt{implies}\}$, $c : \mathtt{Op}$.
  - $\mathtt{cte} : \mathtt{Bool} \to \mathtt{Prop}$.
  - $\mathtt{var} : \mathtt{Pos} \to \mathtt{Prop}$.
  - $\mathtt{neg} : \mathtt{Prop} \to \mathtt{Prop}$.
  - $\mathtt{op} : \mathtt{Prop} \times \mathtt{Op} \times \mathtt{Prop} \to \mathtt{Prop}$.

Exo 3.13† Pouvez-vous construire le terme sur Σ_{AST} associé aux mots suivants en notation préfixe :

1. `& & - & 1 - 2 | - 2 3 | t - 3`
2. `& & - & 1 - 2 | - 2 3`

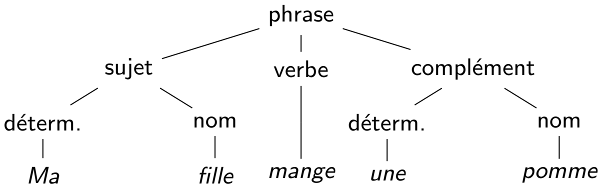
Introduction à l'analyse syntaxique

Analogie avec le français pour lire "*Ma fille mange une pomme*"

1) Analyse lexicale reconnaît nature de chaque *mot*

déterm.	nom	verbe	déterm.	nom
Ma	*fille*	*mange*	*une*	*pomme*

2) Analyse grammaticale reconnaît structure de la *phrase*



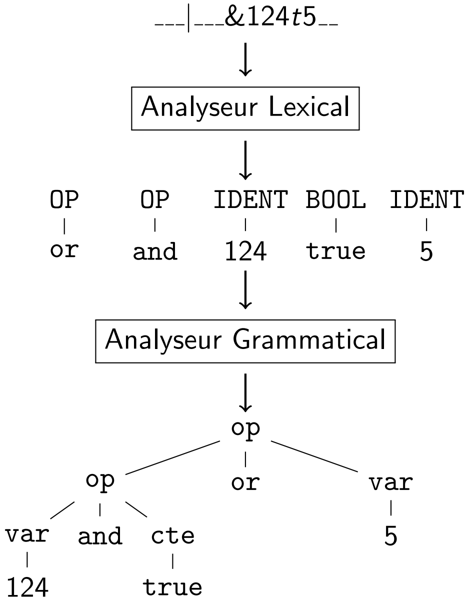
Architecture d'un analyseur syntaxique

$$
\begin{array}{c}
\text{suite de caractères} \\
\downarrow \\
\boxed{\text{Analyseur Lexical}} \\
\downarrow \\
\text{suite de lexèmes suivi d'un éventuel } \perp \\
\downarrow \\
\boxed{\text{Analyseur Grammatical}} \\
\swarrow \qquad\qquad \searrow \\
\text{AST} \qquad\qquad\qquad \perp
\end{array}
$$

lexème = un mot formé d'une suite de caractères

analyse grammaticale = 1 BNF attribuée produisant AST dont chaque *terminal* nomme un *ensemble de lexèmes* (token) (avec ensemble fini de terminaux).

Exemple de construction d'un AST



Principes de l'analyse lexicale

• Lecture du prochain lexème **piloté** par analyseur grammatical.
(origine du mot "*token*").
Intérêt : éviter de stocker toute la suite des lexèmes en mémoire !

• Langage d'entrée inclus dans "$(\mathtt{SEP} \cup \mathtt{TOKEN}_1 \cup \ldots \cup \mathtt{TOKEN}_n)^*$"
où $\mathtt{SEP}$, $\mathtt{TOKEN}_1$, ..., $\mathtt{TOKEN}_n$ sont des langages 2 à 2 disjoints
avec $\mathtt{SEP}$ un langage régulier de "*séparateurs*"
et $\mathtt{TOKEN}_1$, ..., $\mathtt{TOKEN}_n$ langages réguliers par "*nature*" de lexèmes.

• *Lecture "gauche/droite" des lexèmes*
prochain lexème = **+ long préfixe** $\in \mathtt{SEP}^*.(\mathtt{TOKEN}_1 \cup \ldots \cup \mathtt{TOKEN}_n)$
Exemple sur "`124&1`", lire "`124`" comme un seul lexème.

$\Rightarrow$ analyseur lexical = automate fini.

Exemple de lexicographie pour **Prop**

Séparateurs $\texttt{SEP} \stackrel{def}{=} \{\text{‘\_’}\}$

Tokens
Choix d'implémentation : chaque sorte externe Σ_{AST} correspond à un type de lexème !

- $\texttt{BOOL} \stackrel{def}{=} \{\text{‘}\texttt{t}\text{’}, \text{‘}\texttt{f}\text{’}\}$
- $\texttt{OP} \stackrel{def}{=} \{\text{‘}\texttt{\&}\text{’}, \text{‘}\texttt{|}\text{’}, \text{‘}\texttt{>}\text{’}\}$
- $\texttt{IDENT} \stackrel{def}{=} \{\text{‘}\texttt{1}\text{’} \ldots \text{‘}\texttt{9}\text{’}\}.\{\text{‘}\texttt{0}\text{’} \ldots \text{‘}\texttt{9}\text{’}\}^*$
- $\texttt{NEG} \stackrel{def}{=} \{\text{‘}\texttt{-}\text{’}\}$

Exo 3.14† Calculer la suite des tokens pour :

1. `123&27|-4_t_6_5_`
2. `___tx2`

L'analyseur grammatical pour **Prop** en notation préfixe

Terminal de la BNF = Token
avec pour non-singleton, valeur du lexème pour attribut

$$\begin{array}{ll} \mathtt{BOOL}\!\uparrow\!\{\mathtt{true}, \mathtt{false}\} & \mathtt{IDENT}\!\uparrow\!\mathbb{N}\setminus\{0\} \\ \mathtt{NEG} & \mathtt{OP}\!\uparrow\!\{\mathtt{and}, \mathtt{or}, \mathtt{implies}\} \end{array}$$

Attribut synthétisé r : mot dans $T_{\Sigma_{AST}}$.

$$\begin{array}{rclll} \mathrm{Parse}\!\uparrow\! r & ::= & \mathtt{BOOL}\!\uparrow\! b & r := & \mathtt{cte}\ b \\ & | & \mathtt{IDENT}\!\uparrow\! i & r := & \mathtt{var}\ i \\ & | & \mathtt{NEG}\ \ \mathrm{Parse}\!\uparrow\! p & r := & \mathtt{neg}\ p \\ & | & \mathtt{OP}\!\uparrow\! o\ \ \mathrm{Parse}\!\uparrow\! p_1\ \ \mathrm{Parse}\!\uparrow\! p_2 & r := & \mathtt{op}\ p_1\ o\ p_2 \end{array}$$

Exo 3.15† définir la signature Σ_{Parse} correspondant à cette BNF (les attributs sur les terminaux seront représentés via des sortes externes).
Quel mot de $T_{\Sigma_{Parse}}$ représente "`& - & 123 - 2 | - 2 3`" ?
NB : cette BNF attribuée décrit une application de $T_{\Sigma_{Parse}} \to T_{\Sigma_{AST}}$.

L'analyseur lexical comme "itérateur" en C

```
typedef enum { BOOL, OP, IDENT, NEG, END } Token;

typedef enum { true, false } Bool;
typedef enum { and, or, implies } Op;

typedef union {
   Bool bool;
   Op   op;
   int  ident;
} Attr;

/* retourne END si plus de lexème
   modifie v->bool ssi retourne BOOL
   modifie v->op ssi retourne OP
   modifie v->ident ssi retourne IDENT
*/
Token next(Attr *v);
```

Ici : cadre simplifié avec arrêt immédiat en cas d'erreur.
En général : un token spécial ERR pour laisser analyseur grammatical gérer erreur.

Méthode gale de construction de l'implémentation

Principes

- langage "$\mathtt{SEP}^*.(\mathtt{TOKEN}_1 \cup \ldots \cup \mathtt{TOKEN}_\mathtt{n})$" régulier donc reconnaissable par automate fini déterministe complet (avec état erreur) A.
- Calcul du prochain lexème :
  1. lecture successive des caractères jusqu'à état erreur dans A (ou fin du texte).
  2. si *état final* recontré, le lexème correspond au dernier état final.
  3. sinon, erreur (pas de lexème).
- Utilisation d'un "*buffer de pré-vision*" (look-ahead) mémorisant caractères lus entre état final et état erreur (préfixe du prochain lexème).

NB Très souvent un buffer de 1 caractère suffit !
Contre-exemple : < et <=> lexèmes, mais pas <=

Autres tâches souvent dédiées à l'analyseur lexical

Pré-processing (limité à des calculs sur langages réguliers)

- suppression des commentaires (considérés comme séparateurs).
- mécanisme de `#include` à la C.

Dictionnaire des identificateurs & mots-clés

- codage efficace des identificateurs dans AST (via "pointeur" partagé pour identificateurs de même nom).
- simplification de l'automate pour discrimination *mot-clés*/*identificateurs*.

SdD de localisation dans le source (pour messages d'erreurs).

L'analyseur grammatical en C (avec Prop $\stackrel{def}{=} T_{\Sigma_{AST}}$)

Principe fonction `parse_rec` basée sur lemme du préfixe unique.

```
static Prop parse_rec() {
    Attr v;
    Token current=next(&v);
    switch (current) {
    case BOOL: return cte(v.bool);
    case IDENT: return var(v.ident);
    case NEG:
         Prop p = parse_rec();
         return neg(p);
    case OP:
         Prop p1 = parse_rec();
         Prop p2 = parse_rec();
         return op(p1,v.op,p2);
    default: // cas du END (ou lexeme inconnu)
         error_unexpected(current, v);
    }
}
```

NB ds pg principal, après appel à "`parse_rec()`", vérifier "`next()==END`" !

Notion d'arbre d'analyse (parse tree)

- On a deux notions d'arbres :
  1. ceux de $T_{\Sigma_{AST}}$ = AST = arbres abstraits
  2. et ceux de $T_{\Sigma_{Parse}}$, appelés *arbres d'analyse* ou *parse trees* en anglais.

- Les arbres de $T_{\Sigma_{Parse}}$ ne sont pas explicitement construits par l'analyseur `parse_rec`. En fait, ils correspondent aux *arbres des appels récursifs* de cette fonction.

Exo 3.16† dessiner l'arbre des appels récursifs de `parse_rec` sur le mot "`& - & 123 - 2 | - 2 3`".