

## Projet 2: SAT

**Daniel Hirschhoff, Olga Kupriianova, Antoine Plet**

<http://perso.ens-lyon.fr/daniel.hirschhoff/P2>

## Projet2 — présentation

- implémenter plusieurs versions d'un **solveur SAT**
- dans le langage de votre choix parmi  
C   Caml   Java
- modalités
  - en binôme
  - appariements par “niveaux proches en programmation”
  - 4 échéances au long du semestre
- séances en salle machines, quelques séances en amphi

# Évaluation

- ▶ capacité à développer du code
  - ▶ de façon réfléchie (clarté, modularité, efficacité)
  - ▶ de manière organisée (échéances, respect des consignes)
- ▶ exigences
  - ▶ adaptées à votre niveau d'expertise
  - ▶ uniformité sur l'organisation
- ▶ travail important, tout au long du semestre
  - ▶ entre les séances notamment (maison, salles machines)
- ▶ petit rapport et petite soutenance, à la fin

*SAT et DPLL*

# Le problème SAT : description, notations

- ▶ on cherche à satisfaire  
une formule logique en **forme normale conjonctive**

$$(x_1 \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_2) \wedge (x_2 \vee x_3 \vee x_5) \wedge (\overline{x_3} \vee x_5)$$

quatre étages:

- ▶ **variables**  $x_1, x_2, \dots, x_k$
- ▶ **litéraux**  $\alpha, \beta$   
exemples:  $x_1, \overline{x_1}, x_2, \overline{x_2}, \dots$        $\overline{x_1}$  : "non  $x_1$ " (parfois  $\neg x_1$ )  
si  $\alpha = \overline{x_7}$ ,  $\alpha$  vrai signifie  $x_7$  faux
- ▶ **clauses**  $C = \alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_n$
- ▶ **formule**  $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_r$
- ▶ **formule satisfiable**: il existe une *assignment* d'une valeur de vérité aux variables telle que chaque clause contienne au moins un littéral vrai
- ▶ cas limite:
  - formule vide: satisfiable
  - clause vide: insatisfiable



# Positionnement

- **satisfiabilité**: rôle central en théorie de la complexité
- des **solveurs**: utiliser la machine pour démontrer des résultats un domaine de recherche:

des méthodes de toutes sortes...

une technologie qui s'affine depuis des décennies

- . sur l'ensemble du spectre, de méthodes complètes à des heuristiques
- . de l'outil-exemple pour chercheur/théoricien à la dimension industrielle

... pour résoudre maints problèmes

logistique, planification, vérification de matériel et de logiciel, jeux, ...

# Positionnement

- **satisfiabilité**: rôle central en théorie de la complexité
- des **solveurs**: utiliser la machine pour démontrer des résultats un domaine de recherche:

des méthodes de toutes sortes...

une technologie qui s'affine depuis des décennies

- . sur l'ensemble du spectre, de méthodes complètes à des heuristiques
- . de l'outil-exemple pour chercheur/théoricien à la dimension industrielle

... pour résoudre maints problèmes

logistique, planification, vérification de matériel et de logiciel, jeux, ...

- ce cours

- un peu : se faire de la culture sur SAT

(est-ce vraiment un sujet de L3?) / "démystifier NP"

**surtout :**

SAT est un support pour apprendre à programmer/s'organiser, dans le cadre d'un projet logiciel non rikiki

*(en particulier, il ne s'agit pas de faire du "SAT ultra sophistiqué")*

# L'algorithme au cœur du solveur: DPLL

Davis-Putnam-Logemann-Loveland, 1962

- ▶ on explore l'espace des affectations possibles des variables  
essayer avec  $x_1 = \text{vrai}$  ...  
    essayer avec  $x_2 = \text{vrai} \dots$  ... **récurivement**  
**si ça ne marche pas,**  
    essayer avec  $x_2 = \text{faux} \dots$  ...



# L'algorithme au cœur du solveur: DPLL

Davis-Putnam-Logemann-Loveland, 1962

- ▶ on explore l'espace des affectations possibles des variables  
essayer avec  $x_1 = \text{vrai}$  ...  
essayer avec  $x_2 = \text{vrai}$ ... ...**récurivement**  
**si ça ne marche pas,**  
essayer avec  $x_2 = \text{faux}$ ... ...
- ▶ exploite ce que l'on appelle le *variable splitting*
  - ▶ on explore toutes les instanciations possibles (*des variables*)
  - ▶ on manipule une *instanciation partielle*, que l'on étend tant qu'elle est consistante (pas de *conflit*)
  - ▶ en cas de conflit, on élimine un ensemble d'instanciations
- ▶ on fait des **paris**

# L'algorithme au cœur du solveur: DPLL

Davis-Putnam-Logemann-Loveland, 1962

- ▶ on explore l'espace des affectations possibles des variables  
essayer avec  $x_1 = \text{vrai}$  ...  
essayer avec  $x_2 = \text{vrai}$ ... ... **récurivement**  
**si ça ne marche pas,**  
essayer avec  $x_2 = \text{faux}$ ... ...
- ▶ exploite ce que l'on appelle le *variable splitting*
  - ▶ on explore toutes les instanciations possibles (*des variables*)
  - ▶ on manipule une *instanciation partielle*, que l'on étend tant qu'elle est consistante (pas de *conflit*)
  - ▶ en cas de conflit, on élimine un ensemble d'instanciations
- ▶ on fait des **paris**  
à chaque pari, on **déduit des conditions nécessaires**  
p.ex. si  $x_3 = \text{vrai}$ , la clause  $\overline{x_3} \vee \overline{x_{12}}$  implique  $x_{12} = \text{faux}$   
↪ on n'énumère ainsi pas *toutes* les affectations possibles

# Algorithme DPLL — étapes essentielles

## 1. *boolean constraint propagation*

déduire

### 1.1 trouver les affectations *nécessaires*

- ▶ si une clause est  $\{\alpha\}$ , alors  $\alpha$  est nécessairement satisfait  
(i.e.,  $x = \text{vrai}$  si  $\alpha = x$ ,  $= \text{faux}$  si  $\alpha = \bar{x}$ )
- ▶ si une variable  $x$  n'apparaît qu'avec une seule polarité,  
déduire sa valeur

### 1.2 propagation des valeurs

si nécessairement  $\alpha$  est satisfait,

- . “éliminer” toutes les clauses contenant  $\alpha$
- . “éliminer”  $\bar{\alpha}$  de toutes les clauses le contenant

### 1.3 recommencer en 1.1

# Algorithme DPLL — étapes essentielles

## 1. *boolean constraint propagation*

déduire

### 1.1 trouver les affectations *nécessaires*

- ▶ si une clause est  $\{\alpha\}$ , alors  $\alpha$  est nécessairement satisfait (i.e.,  $x = \text{vrai}$  si  $\alpha = x$ ,  $= \text{faux}$  si  $\alpha = \bar{x}$ )
- ▶ si une variable  $x$  n'apparaît qu'avec une seule polarité, déduire sa valeur

### 1.2 propagation des valeurs

si nécessairement  $\alpha$  est satisfait,

- . “éliminer” toutes les clauses contenant  $\alpha$
- . “éliminer”  $\bar{\alpha}$  de toutes les clauses le contenant

### 1.3 recommencer en 1.1

## 2. *choix*

décider

*choisir* une variable  $x_i$  encore inconnue,  
et lui attribuer une valeur (vrai ou faux)

# Algorithme DPLL — étapes essentielles

## 1. *boolean constraint propagation*

déduire

### 1.1 trouver les affectations *nécessaires*

- si une clause est  $\{\alpha\}$ , alors  $\alpha$  est nécessairement satisfait (i.e.,  $x = \text{vrai}$  si  $\alpha = x$ ,  $= \text{faux}$  si  $\alpha = \bar{x}$ )
- si une variable  $x$  n'apparaît qu'avec une seule polarité, déduire sa valeur

### 1.2 propagation des valeurs

si nécessairement  $\alpha$  est satisfait,

- . "éliminer" toutes les clauses contenant  $\alpha$
- . "éliminer"  $\bar{\alpha}$  de toutes les clauses le contenant (★)

### 1.3 recommencer en 1.1

## 2. *choix*

décider

*choisir* une variable  $x_i$  encore inconnue,  
et lui attribuer une valeur (vrai ou faux)

## 3. *backtrack*

rebrousser chemin

l'étape (★) peut engendrer une clause vide: **conflit**  
on revient alors sur le dernier *choix* ayant été fait

(choix, pas déduction)

## DPLL – exemple

autres transparents

# Paris et backtrack

- à un instant donné dans l'exécution de DPLL, on a parié sur la valeur d'un certain nombre de variables
- on maintient une **pile de tranches**
  - on a fait  $n$  **paris** sur des variables  $x_{c_1}, \dots, x_{c_n}$
  - chaque choix a entraîné, par BCP, l'assignation d'un certain nombre de variables (elles sont dans la même tranche)  
$$x_{c_1}, y_{c_1}^1, \dots, y_{c_1}^{i_1} \mid x_{c_2}, y_{c_2}^1, \dots, y_{c_2}^{i_2}, \dots \mid x_{c_n}, y_{c_n}^1, \dots, y_{c_n}^{i_n}$$
- lorsque l'on rebrousse chemin, il faut
  - retourner sa veste pour le dernier pari *où cela est possible*
    - si on a parié sur  $x_{c_n} = \text{faux}$  car on avait déjà vu que  $x_{c_n} = \text{vrai}$  était contradictoire, remonter à  $x_{c_{n-1}}$
    - *(dans une telle situation, on peut imaginer que  $x_{c_n} = \text{faux}$  n'était pas un pari, mais une **déduction**)*
  - annuler l'affectation des variables ayant découlé de ce dernier pari

# Pensez avant de programmer

- choix des structures de données
  - qu'est-ce qu'une variable, un littéral, une clause ?  
(dans votre programme)
  - phase de propagation
    - à quoi veut-on accéder et comment?
      - . toutes les clauses contenant telle variable
      - . le nombre de littéraux "vivants" dans telle clause
      - . le nombre de clauses "vivantes"
    - quelles opérations faut-il être en mesure d'effectuer?
  - l'affectation courante
    - que faire quand on revient sur l'affectation d'une variable?



# Pensez avant de programmer

- ▶ choix des structures de données
  - ▶ qu'est-ce qu'une variable, un littéral, une clause ?  
(dans votre programme)
  - ▶ phase de propagation
    - ▶ à quoi veut-on accéder et comment?
      - . toutes les clauses contenant telle variable
      - . le nombre de littéraux "vivants" dans telle clause
      - . le nombre de clauses "vivantes"
    - ▶ quelles opérations faut-il être en mesure d'effectuer?
  - ▶ l'affectation courante  
que faire quand on revient sur l'affectation d'une variable?
- ▶ n'utilisez pas une grande fonction récursive: implémentez la récursion à la main (on manipule la pile, on a un meilleur contrôle)

# Pensez avant de programmer

- ▶ choix des structures de données
  - ▶ qu'est-ce qu'une variable, un littéral, une clause ?  
(dans votre programme)
  - ▶ phase de propagation
    - ▶ à quoi veut-on accéder et comment?
      - . toutes les clauses contenant telle variable
      - . le nombre de littéraux "vivants" dans telle clause
      - . le nombre de clauses "vivantes"
    - ▶ quelles opérations faut-il être en mesure d'effectuer?
  - ▶ l'affectation courante  
que faire quand on revient sur l'affectation d'une variable?
- ▶ n'utilisez pas une grande fonction récursive: implémentez la récursion à la main (on manipule la pile, on a un meilleur contrôle)
- ▶ autre aspect important : programmez modulairement  
dans un mois, si vous voulez changer une des structures de données, il faudrait que cela ne soit pas une catastrophe  
(+ interdit de dire "désolé, mais je ne prévois pas de me tromper"!) )

# Rendu 1 — DPLL

- ▶ en entrée: une formule logique
- ▶ en sortie: non/oui et une affectation des variables
- ▶ à l'intérieur: algorithme DPLL “simple”
  - ▶ **structure modulaire**, ouvrant la voie à des raffinements successifs
    - un fichier pour la boucle principale
  - ▶ **structures de données** pour
    - ▶ la représentation des variables, littéraux, clauses
      - . une variable peut avoir trois états: inconnu, vrai, faux
    - ▶ l'état courant de la recherche
  - ▶ **efficacité raisonnable**
    - ▶ typiquement, 80-90 % du temps passé dans la propagation des informations
    - ▶ pouvoir au moins soupçonner les sources majeures d'inefficacité
  - ▶ **traitement de l'entrée**
    - ▶ déductions diverses (p.ex. détecter les  $x \vee \bar{x}$ )
    - ▶ éventuellement, tri des variables
    - ▶ éventuellement, collecte d'informations sur les clauses

# Recommandations

- ▶ `minisat` est installé sur les machines des salles libre-service
  - ▶ l'exécution de référence
- ▶ faites des tests en nombre raisonnable
  - ▶ un sous-répertoire avec des fichiers de tests que vous écrivez/engendrez
- ▶ implémentez *ce qui vous est demandé*
- ▶ insistons: importance de la ponctualité des rendus!
  - ▶ les rendus sont incrémentaux, mais ce n'est pas uniquement le résultat en fin de semestre qui compte

Soyons alphabétisés

**Lisez** les sujets de rendu.

## À titre d'information: rendus $n + 1$

- variations sur la manière dont on choisit le prochain pari
  - sur quelle variable miser, avec quelle valeur
- une technique *astucieuse* pour la propagation des unités
- calculer des choses *astucieuses* sur l'état courant de l'exploration
- extension, en cours de route, de l'ensemble des clauses
- backtrack *astucieux* (plus en amont que le dernier pas)
- on décidera de repartir à zéro après  $K$  conflits
- on tirera d'emblée au hasard une affectation pour toutes les variables, pour ensuite touiller de ci de là jusqu'à satisfaire la formule

## À titre d'information: rendus $n + 1$

- variations sur la manière dont on choisit le prochain pari
  - sur quelle variable miser, avec quelle valeur
- une technique *astucieuse* pour la propagation des unités
- calculer des choses *astucieuses* sur l'état courant de l'exploration
- extension, en cours de route, de l'ensemble des clauses
- backtrack *astucieux* (plus en amont que le dernier pas)
- on décidera de repartir à zéro après  $K$  conflits
- on tirera d'emblée au hasard une affectation pour toutes les variables, pour ensuite touiller de ci de là jusqu'à satisfaire la formule

**ne vous effrayez pas: vous pouvez dans un premier temps faire “quelque chose qui marche”, quitte à devoir reprendre une partie de la structure par la suite**

*Transformation de Tseitin*



## Formules quelconques, lois de de Morgan

- la procédure DPLL s'applique à des formules en *forme normale conjonctive* (CNF)

$$(x_1 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_4 \vee x_5) \wedge (\overline{x_2} \vee \overline{x_3})$$

- on veut prendre en entrée une formule logique plus complexe

$$(\neg p \wedge (q \Rightarrow r)) \Rightarrow (q \vee \neg p)$$

# Formules quelconques, lois de de Morgan

- la procédure DPLL s'applique à des formules en *forme normale conjonctive* (CNF)

$$(x_1 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_4 \vee x_5) \wedge (\overline{x_2} \vee \overline{x_3})$$

- on veut prendre en entrée une formule logique plus complexe

$$(\neg p \wedge (q \Rightarrow r)) \Rightarrow (q \vee \neg p)$$

## Lois de de Morgan :

$$[(p \wedge q) \vee r] = (p \vee r) \wedge (q \vee r) \qquad [\neg(p \wedge q)] = \neg p \vee \neg q$$

$$[\neg(p \vee q)] = \neg p \wedge \neg q \qquad [p \Rightarrow q] = \neg p \vee q \qquad [\neg\neg p] = p$$

- on obtient une formule en CNF en itérant ces lois...
- ...mais la distributivité fait exploser la taille
- inévitables si on veut préserver le sens de la formule

## Formules quelconques, lois de de Morgan

- la procédure DPLL s'applique à des formules en *forme normale conjonctive* (CNF)

$$(x_1 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_4 \vee x_5) \wedge (\overline{x_2} \vee \overline{x_3})$$

- on veut prendre en entrée une formule logique plus complexe
- $$(\neg p \wedge (q \Rightarrow r)) \Rightarrow (q \vee \neg p)$$

### Lois de de Morgan :

$$[(p \wedge q) \vee r] = (p \vee r) \wedge (q \vee r) \qquad [\neg(p \wedge q)] = \neg p \vee \neg q$$

$$[\neg(p \vee q)] = \neg p \wedge \neg q \qquad [p \Rightarrow q] = \neg p \vee q \qquad [\neg\neg p] = p$$

- on obtient une formule en CNF en itérant ces lois...  
...mais la distributivité fait exploser la taille
- inévitables si on veut préserver le sens de la formule
- or il suffit de préserver la **satisfiabilité**  
et on veut aussi pouvoir engendrer, le cas échéant, un contre-exemple de la formule de départ

# Transformation de Tseitin, définition

- pour chaque sous-formule  $p$  de la formule de départ, on introduit une nouvelle variable  $\xi_p$
- on y va inductivement, pour associer à chaque sous-formule  $p$  une formule  $[p]$ , *directement en forme normale conjonctive*:

$$[p = p_1 \vee p_2] = (\neg \xi_p \vee \xi_{p_1} \vee \xi_{p_2}) \wedge (\xi_p \vee \neg \xi_{p_1}) \wedge (\xi_p \vee \neg \xi_{p_2})$$

$$[p = p_1 \wedge p_2] = (\neg \xi_p \vee \xi_{p_1}) \wedge (\neg \xi_p \vee \xi_{p_2}) \wedge (\xi_p \vee \neg \xi_{p_1} \vee \neg \xi_{p_2})$$

$$[p = \neg p_1] = (\neg \xi_p \vee \neg \xi_{p_1}) \wedge (\xi_p \vee \xi_{p_1})$$

$$[p = x] = (\neg \xi_p \vee x) \wedge (\xi_p \vee \neg x)$$

ainsi, par exemple, lorsque  $p = p_1 \vee p_2$ ,  $[p]$  exprime que  $p$  est satisfaite si et seulement si  $p_1 \vee p_2$  l'est

- une formule  $p$  est transformée en la conjonction

$$\xi_p \wedge \bigwedge_{p' \text{ sous-formule de } p} [p']$$

le  $\xi_p$  impose que la formule initiale (à la racine) soit satisfaite

# Tseitin, propriétés

on se donne une formule  $p$ , on en calcule la transformée de Tseitin, notée (par abus)  $[p]$

- ▶  $[p]$  se calcule en temps **linéaire** par rapport à la taille de  $p$
- ▶  $p$  et  $[p]$  sont **équi-satisfiables**
  - ▶ toute valuation qui satisfait  $[p]$  satisfait  $p$  (et satisfait toujours  $p$  en modifiant la valeur de variables ne se trouvant pas dans  $p$ )
  - ▶ une valuation qui satisfait  $p$  peut être étendue en une valuation qui satisfait  $[p]$
- ▶ (on peut optimiser la représentation en mémoire en partageant des sous-expressions)

# Retour au rendu 1

- ▶ deux versions de votre solveur
  1. le solveur SAT “brut”  
qui mange des formules en CNF
  2. le “solveur convivial”  
qui mange des formules quelconques
- ▶ 2 applique Tseitin pour s'appuyer sur 1  
cela rappelle (une partie de) la structure d'un *compilateur*

*Très rapidement,  
quelques éléments de compilation*

*analyses lexicale et syntaxique: fabriquer des arbres*

# Interpréter / compiler

- **interprète:** implémentation de la sémantique opérationnelle  
exécuter le programme
- **compilateur:** traduction  
traduire (en préservant le sens)  
(p.ex.  $\text{IMP} \rightsquigarrow \text{assembleur}$ )

interprètes et compilateurs sont des programmes manipulant des programmes



# Un compilateur

- traducteur de code à code (de *fichier source* à *fichier objet*)

- anatomie sommaire 1 → 2 → 3

## 1. front end

du fichier de texte à une représentation arborescente

```
"let x = 3 in (f x)+2"
```

ou plutôt

```
['l';'e';'t';' '; 'x';' '; '='; '3'; ' '; 'i'; 'n'; ' '; ' '; '('; 'f'; ' '; 'x'; ')'; '+'; '2'; '\n']
```

```
Let(Var "x", Cst 3, Add(App(Var "f", Var "x"), Cst 2))
```

- ## 2. des tas de **transformations** (*représentations intermédiaires*)

## 3. back end

génération de code: d'une représentation arborescente à un fichier de texte

```
[Push(rx);Set(rj,f_addr);Call;Pop;Set(r0,2);Add]
```

```
start:  push(rx);  
        set(rj,f_addr);  
        call;  
        pop;  
        set(r0,2);  
        add;
```

- “tout” est dans l'étape **2**: analyses, transformations, réécritures, algorithmique, optimisations, ...

# Les deux étapes dans le front end

## ▸ analyse lexicale

flot de caractères (source) → flot de *lexèmes*

- lexème (*token*): “atome” du langage
- typiquement:
  - mots-clefs (`let`, `begin`, `while`, ...)
  - symboles réservés (`(`, `+`, `;;`, `;`, ...)
  - identificateurs (`f`, `toto`, ...)

ainsi `32*52+(let x = 5 in x*x)`

→ `INT(32)`, `MULT`, `INT(52)`, `ADD`, `LPAREN`, `LET`, `ID("x")`, `EGAL`, `INT(5)`,  
`IN`, `ID("x")`, `MULT`, `ID("x")`, `RPAREN`  
(`INT` et `ID` ont un *attribut*, entier et chaîne de caractères respectivement)

# Les deux étapes dans le front end

## ▸ analyse lexicale

flot de caractères (source) → flot de *lexèmes*

- lexème (*token*): “atome” du langage
- typiquement:
  - mots-clefs (`let`, `begin`, `while`, ...)
  - symboles réservés (`(`, `+`, `;;`, `;`, ...)
  - identificateurs (`f`, `toto`, ...)

ainsi `32*52+(let x = 5 in x*x)`

→ `INT(32), MULT, INT(52), ADD, LPAREN, LET, ID("x"), EGAL, INT(5), IN, ID("x"), MULT, ID("x"), RPAREN`  
(`INT` et `ID` ont un *attribut*, entier et chaîne de caractères respectivement)

## ▸ analyse syntaxique

flot de lexèmes → *arbre de syntaxe abstraite*

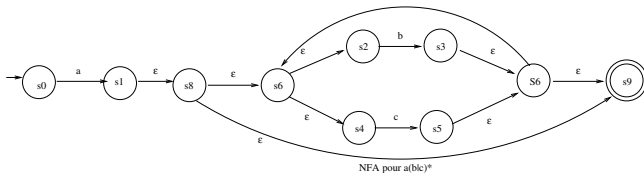
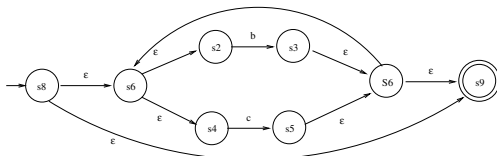
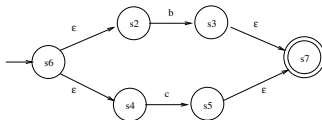
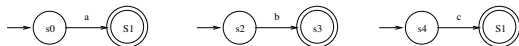
→ `Add( Mult(Int(32), Int(52)),  
Let("x", Int(5), Mult(Var("x"), Var("x")))) )`

- étape intermédiaire: arbre d'analyse syntaxique (*parse tree*)

# Analyse lexicale

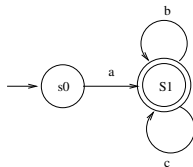
- chaque lexème est décrit par une *expression régulière*
- principaux éléments (syntaxe de `ocamllex`):
  - caractère '\$', chaîne de caractères "else"
  - intervalle `['0'-'9']` (*un chiffre*)
  - disjonction (de caractères)  
`['\t' ' ']` (*tabulation ou espace*)
  - juxtaposition `['A'-'Z']['a'-'z' 'A'-'Z']`  
(*mot de 2 lettres commençant par une majuscule*)
  - répétitions: + signifie au moins 1, \* zéro ou plus  
`['a'-'z']+['a'-'z' '0'-'9']*`  
(*ça commence par une lettre puis des lettres ou des chiffres*)
  - disjonction `a* | b*`
- en sortie de l'analyse lexicale: des *mots*

# Expression régulière $\leftrightarrow$ automate non déterministe



# Déterminisation, minimisation

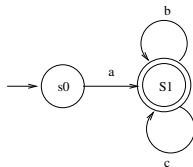
- à partir de l'automate du transparent précédent, on dispose de procédures pour *déterminiser* l'automate (explosion du nombre d'états), puis le *minimiser*



- on aboutit à

# Déterminisation, minimisation

- à partir de l'automate du transparent précédent, on dispose de procédures pour *déterminiser* l'automate (explosion du nombre d'états), puis le *minimiser*



- on aboutit à

- comment implémenter l'automate résultant?

- une table (très creuse)

état	a	b	c	d	e
e1	-	e2	e3	-	-
e2	e4	-	-	-	-
e3	-	-	e3	-	-

- éliminer les états: un plat de spaghetti, fait de *if* et de *goto*

# Analyse lexicale : au total

- ▶ on décrit le **dictionnaire**

ensemble d'expressions régulières,  
auxquelles on associe un nom (*avec éventuellement un attribut*)  
un lexème

"let"	{ LET }
"in"	{ IN }
['0' - '9']+ as s	{ INT (int_of_string s) }

- ▶ “magiquement”, on obtient un programme qui reconnaît les mots du dictionnaire (et proteste sinon)

DÉMO



# Analyse syntaxique

- l'analyse syntaxique se fonde sur une approche plus puissante:

## **règles de grammaire**

- les règles de grammaire font intervenir les lexèmes et des “variables” (les non terminaux)
- exemple de grammaire:

$E ::= K \mid E + E \mid E * E \mid (E) \mid \text{let } Id = E \text{ in } E$

- $E$  non terminal (il peut y en avoir plusieurs)
- $K, \text{let}, Id, +, *, (, ), \text{in}, =$  lexèmes

présentation alternative:

$E \rightarrow K$        $E \rightarrow E + E$        $E \rightarrow E * E$        $E \rightarrow (E)$        $E \rightarrow \text{let } Id = E \text{ in } + E$

# Analyse syntaxique

- l'analyse syntaxique se fonde sur une approche plus puissante:

## règles de grammaire

- les règles de grammaire font intervenir les lexèmes et des "variables" (les non terminaux)
- exemple de grammaire:

$E ::= K \mid E + E \mid E * E \mid (E) \mid \text{let } Id = E \text{ in } E$

- $E$  non terminal (il peut y en avoir plusieurs)
- $K, \text{let}, Id, +, *, (, ), \text{in}, =$  lexèmes

présentation alternative:

$E \rightarrow K$        $E \rightarrow E + E$        $E \rightarrow E * E$        $E \rightarrow (E)$        $E \rightarrow \text{let } Id = E \text{ in } + E$

- analyse lexicale : du **flot** de caractères au **flot** de lexèmes
- analyseur syntaxique (ou *parser*) : applique les règles de grammaire pour *reconnaître* une suite de lexèmes
  - on change la structure : un **flot** (de lexèmes) devient un **arbre**
  - on construit des *phrases* à partir de *mots*

# Reconnaître une séquence de lexèmes

- l'idée est de construire un *arbre de dérivation* permettant de reconnaître le flot de lexèmes
  - grammaire :  $E ::= E + E \mid E * E \mid K$      $K$  un entier
  - soit le flot `32, +, 26, *, 2`

# Reconnaître une séquence de lexèmes

- l'idée est de construire un *arbre de dérivation* permettant de reconnaître le flot de lexèmes

- grammaire :  $E ::= E + E \mid E * E \mid K$      $K$  un entier

- soit le flot **32, +, 26, \*, 2**

- on peut reconnaître de deux manières:

$E$	$\rightarrow$	$E + E$		$E$	$\rightarrow$	$E * E$
	$\rightarrow$	$K_{32} + E$			$\rightarrow$	$E * K_2$
	$\rightarrow$	$K_{32} + E * E$	ou alors		$\rightarrow$	$E + E * K_2$
	$\rightarrow\rightarrow$	$K_{32} + K_{26} * K_2$			$\rightarrow\rightarrow$	$K_{32} + K_{26} * K_2$

deux arbres différents: *ambiguïté*

# Reconnaître une séquence de lexèmes

- l'idée est de construire un *arbre de dérivation* permettant de reconnaître le flot de lexèmes

- grammaire :  $E ::= E + E \mid E * E \mid K$      $K$  un entier

- soit le flot **32, +, 26, \*, 2**

- on peut reconnaître de deux manières:

$E$	$\rightarrow$	$E + E$		$E$	$\rightarrow$	$E * E$
	$\rightarrow$	$K_{32} + E$			$\rightarrow$	$E * K_2$
	$\rightarrow$	$K_{32} + E * E$	ou alors		$\rightarrow$	$E + E * K_2$
	$\rightarrow\rightarrow$	$K_{32} + K_{26} * K_2$			$\rightarrow\rightarrow$	$K_{32} + K_{26} * K_2$

deux arbres différents: *ambiguïté*

- aucun moyen en revanche de reconnaître **9 \* 1 + + 1**

## Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c$	shift

## Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c\$$	shift
$\$a$	$+b * c\$$	reduce : $E \rightarrow a$

## Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c$ \$	shift
\$a	$+ b * c$ \$	reduce : $E \rightarrow a$
\$E	$+ b * c$ \$	shift



## Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c \$$	shift
$\$a$	$+ b * c \$$	reduce : $E \rightarrow a$
$\$E$	$+ b * c \$$	shift
$\$E+$	$b * c \$$	shift

## Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c \$$	shift
$\$a$	$+ b * c \$$	reduce : $E \rightarrow a$
$\$E$	$+ b * c \$$	shift
$\$E +$	$b * c \$$	shift
$\$E + b$	$* c \$$	reduce : $E \rightarrow b$

# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c \$$	shift
$\$a$	$+ b * c \$$	reduce : $E \rightarrow a$
$\$E$	$+ b * c \$$	shift
$\$E +$	$b * c \$$	shift
$\$E + b$	$* c \$$	reduce : $E \rightarrow b$
$\$E + E$	$* c \$$	shift <b>(très malin)</b>

# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c \$$	shift
$\$a$	$+ b * c \$$	reduce : $E \rightarrow a$
$\$E$	$+ b * c \$$	shift
$\$E +$	$b * c \$$	shift
$\$E + b$	$* c \$$	reduce : $E \rightarrow b$
$\$E + E$	$* c \$$	shift <b>(très malin)</b>
$\$E + E *$	$c \$$	shift

# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c$ \$	shift
\$a	$+ b * c$ \$	reduce : $E \rightarrow a$
\$E	$+ b * c$ \$	shift
\$E+	$b * c$ \$	shift
\$E + b	$* c$ \$	reduce : $E \rightarrow b$
\$E + E	$* c$ \$	shift (très malin)
\$E + E*	$c$ \$	shift
\$E + E * c	\$	reduce : $E \rightarrow c$

# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c$ \$	shift
\$a	$+ b * c$ \$	reduce : $E \rightarrow a$
\$E	$+ b * c$ \$	shift
\$E+	$b * c$ \$	shift
\$E + b	$* c$ \$	reduce : $E \rightarrow b$
\$E + E	$* c$ \$	shift (très malin)
\$E + E*	$c$ \$	shift
\$E + E * c	\$	reduce : $E \rightarrow c$
\$E + E * E	\$	reduce : $E \rightarrow E * E$

# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c \$$	shift
$\$a$	$+ b * c \$$	reduce : $E \rightarrow a$
$\$E$	$+ b * c \$$	shift
$\$E +$	$b * c \$$	shift
$\$E + b$	$* c \$$	reduce : $E \rightarrow b$
$\$E + E$	$* c \$$	shift (très malin)
$\$E + E *$	$c \$$	shift
$\$E + E * c$	$\$$	reduce : $E \rightarrow c$
$\$E + E * E$	$\$$	reduce : $E \rightarrow E * E$
$\$E + E$	$\$$	reduce : $E \rightarrow E + E$

# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c$ \$	shift
\$a	$+ b * c$ \$	reduce : $E \rightarrow a$
\$E	$+ b * c$ \$	shift
\$E+	$b * c$ \$	shift
\$E + b	$* c$ \$	reduce : $E \rightarrow b$
\$E + E	$* c$ \$	shift (très malin)
\$E + E*	$c$ \$	shift
\$E + E * c	\$	reduce : $E \rightarrow c$
\$E + E * E	\$	reduce : $E \rightarrow E * E$
\$E + E	\$	reduce : $E \rightarrow E + E$
\$E	\$	accept



# Ce que fait le parser

$E ::= E + E \mid E * E \mid (E) \mid a \mid b \mid c$        $a+b*c$

pile	entrée	action
\$	$a + b * c \$$	shift
$\$a$	$+ b * c \$$	reduce : $E \rightarrow a$
$\$E$	$+ b * c \$$	shift
$\$E +$	$b * c \$$	shift
$\$E + b$	$* c \$$	reduce : $E \rightarrow b$
$\$E + E$	$* c \$$	shift (très malin)
$\$E + E *$	$c \$$	shift
$\$E + E * c$	$\$$	reduce : $E \rightarrow c$
$\$E + E * E$	$\$$	reduce : $E \rightarrow E * E$
$\$E + E$	$\$$	reduce : $E \rightarrow E + E$
$\$E$	$\$$	accept

- à la fin, on a un arbre  $\text{add}(\text{id}(a), \text{mul}(\text{id}(b), \text{id}(c)))$

# Lex & Yacc, Flex & Bison, ...

- analyse syntaxique : on écrit la grammaire, et on associe à chaque règle une *action sémantique* (*construction de l'arbre*)
- DÉMO avec `ocamllex` `ocamlyacc`

# Lex & Yacc, Flex & Bison, ...

- ▶ analyse syntaxique : on écrit la grammaire, et on associe à chaque règle une *action sémantique* (*construction de l'arbre*)
- ▶ DÉMO avec `ocamllex` `ocamlyacc`
- ▶ *remarque* : avec yacc, on ôte les ambiguïtés en “bricolant”, pas en réécrivant la grammaire (comme en FDI)
- ▶ on trouve “*partout*” les outils pour les analyses lexicale et syntaxique

```

%{
(* ——— preamble: ici du code Caml ——— *)

open Expr  (* rappel: dans expr.ml:
            type expr = Const of int | Add of expr*expr | Mull of expr*expr *)

%}
/* description des lexemes */

%token <int> INT      /* le lexeme INT a un attribut entier */
%token PLUS TIMES
%token LPAREN RPAREN
%token EOL            /* retour a la ligne */

%left PLUS
%left TIMES

%start main           /* "start" signale le point d'entree: c'est ici main */
%type <Expr.expr> main /* on _doit_ donner le type du point d'entree */

%%
/* ——— debut des regles de grammaire ——— */
/* a droite, les valeurs associees */
main:
  expr EOL           { $1 } /* on veut reconnaitre un "expr" */
;
expr: /* regles de grammaire pour les expressions */
  | INT              { Const $1 }
  | LPAREN expr RPAREN { $2 } /* on recupere le deuxieme element */
  | expr PLUS expr   { Add($1,$3) }
  | expr TIMES expr   { Mul($1,$3) }
;

```

```

%{ // useful functions.
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream> //for dag output
#include "ast.h"

using namespace std;

int line_number = 1; /* number of current source line */
extern int yylex(); /* lexical analyzer generated from lex.l */
extern char *yytext; /* last token, defined in lex.l
*/

void yyerror(char *){
printf(stderr, "line_%d:_syntax_error._Last_token_was_\"%s\"\n", line_number, yytext);
exit(1);
}

void error(char *){
fprintf(stderr, "line_%d:_error:_%s\n", line_number, s);
/* Axiom */
%start expr
}

struct expr *parsing_result = NULL;
}%

//type of non terminals
%union {
    double number;
    char* id_string;
    struct expr *expr;
}

//token declaration for minic input
%token TK_PLUS TK_MINUS TK_MUL TK_DIV
%token TK_NUM TK_VAR
%token TK_LPAR TK_RPAR

/* Associativity */
%left TK_PLUS TK_MINUS
%left TK_MUL TK_DIV

%type<number> TK_NUM
%type<id_string> TK_VAR
%type<expr> e_expr t_expr f_expr

```

```

/* Axiom */
%start expr

%%

//ETF (sub-)grammar

e_expr:
e_expr TK_PLUS t_expr { $$ = expr_binop($1, $3, PLUS) ;}
| e_expr TK_MINUS t_expr { $$ = expr_binop($1, $3, MINUS) ;}
| t_expr
;

t_expr:
t_expr TK_MUL f_expr { $$ = expr_binop($1, $3, MULT) ;}
| t_expr TK_DIV f_expr { $$ = expr_binop($1, $3, DIV) ;}
| f_expr
;

f_expr:
TK_NUM { $$ = expr_number($1); }
| TK_VAR { $$ = expr_var($1); }
| TK_LPAR e_expr TK_RPAR { $$ = $2; }
;

expr:
e_expr { parsing_result = $1; }
;

```

# Les deux semaines à venir

## ▸ la semaine prochaine

- TP: analyses lexicale et syntaxique  
Tseitin

*tout le monde commence le projet, seul(e)*

- constitution des binômes

1. **questionnaire**

2. stratification

<http://perso.ens-lyon.fr/daniel.hirschkoef/P2>

3. vous m'envoyez un mail

*"nous sommes X et Y, et nous choisissons le langage Z"*

contrainte : X et Y distants d'au plus un "niveau" dans la stratification

## ▸ la semaine suivante

- démarrage du projet (les binômes sont constitués)

## ▸ premier rendu

- pour le **????** *sans doute le 16 février*

*Améliorer DPLL*

*Watched literals*

*(littéraires surveillés)*





# Watched literals

- ▶ une technique permettant d'économiser du temps lors de la propagation des contraintes
  - ▶ *“Chaff: Engineering an Efficient SAT Solver”*, Moskewicz, Madigan, Zhao, Zhang, Malik, DAC 2001.
  - ▶ questions de droits: technique brevetée ?

# Watched literals

- ▶ une technique permettant d'économiser du temps lors de la propagation des contraintes
  - ▶ *"Chaff: Engineering an Efficient SAT Solver"*, Moskewicz, Madigan, Zhao, Zhang, Malik, DAC 2001.
  - ▶ questions de droits: technique brevetée ?
- ▶ remarque:  
les clauses ont toutes au moins deux littéraux
- ▶ *propagation* lorsque tous les littéraux sauf un sont à faux



## Watched literals — principes

- ▶ pour chaque clause, on ne regarde que deux littéraux...  
...qui ne sont pas à faux (vrai ou inconnu)
  - ▶ si c'est impossible, c'est que la clause en question est inactive  
autrement dit, un littéral est à vrai



# Watched literals — principes



- ▶ pour chaque clause, on ne regarde que **deux littéraux**...  
... qui ne sont pas à faux (vrai ou inconnu)
    - ▶ si c'est impossible, c'est que la clause en question est inactive  
autrement dit, un littéral est à vrai
  - ▶ lorsque le littéral  $\alpha$  est mis à faux,
    - ▶ pour toutes les clauses  $C$  où  $\alpha$  est surveillé, on cherche un autre littéral à surveiller
      - ▶ si pas possible, déclenchement de propagation supplémentaire
- NB: là où  $\alpha$  apparaît sans être surveillé, *on ne fait rien*
- ▶ *optionnel*: pour toutes les clauses  $C$  contenant  $\bar{\alpha}$  (surveillé ou non),
    - ▶ on se fait la remarque que  $C$  est satisfaite
    - ▶ si on vient de rendre la clause satisfiable,  
on installe  $\bar{\alpha}$  comme littéral surveillé (priorité aux littéraux vrais par rapport aux inconnus)
      - ↔ on détermine plus facilement si une clause est satisfaite

## Watched literals — commentaires

- ▶ alors donc :
  - ▶ on ne regarde qu'au plus deux littéraux pour savoir si une clause est satisfaite
  - ▶ invariant: tant que la clause n'est pas satisfaite, aucun des deux littéraux n'est à faux
  - ▶ ce faisant, on regarde moins de pointeurs  
*" $\alpha$  apparaît-il dans  $C_i$ ?"* vs *" $\alpha$  est-il surveillé dans  $C_i$ ?"*

## Watched literals — commentaires

- ▶ alors donc :
  - ▶ on ne regarde qu'au plus deux littéraux pour savoir si une clause est satisfaite
  - ▶ invariant: tant que la clause n'est pas satisfaite, aucun des deux littéraux n'est à faux
  - ▶ ce faisant, on regarde moins de pointeurs  
*" $\alpha$  apparaît-il dans  $C_i$ ?" vs " $\alpha$  est-il surveillé dans  $C_i$ ?"*
- ▶ backtrack: on laisse les littéraux surveillés inchangés!  
*(on ne bouge pas les jumelles)*
  - ▶ les jumelles ne doivent pas revenir à leur position antérieure
  - ▶ l'état n'est pas nécessairement le même, mais essentiellement équivalent
  - ▶ remarque: si on surveille (**faux**, -), alors, par l'invariant, on surveille (**faux**, vrai), et lors d'un backtrack on passera de **faux** à ? avant de modifier les littéraux non surveillés de la clause
- ▶ particulièrement efficace sur les entrées ayant de longues clauses
- ▶ NB : avec WL, pas de propagation "littéral pur"