

Compilation

Cours n°8: Linéarisation – Réalisation des trames de pile
De Pseudo-Pascal vers C ou Java
Ramassage de miettes – Exceptions

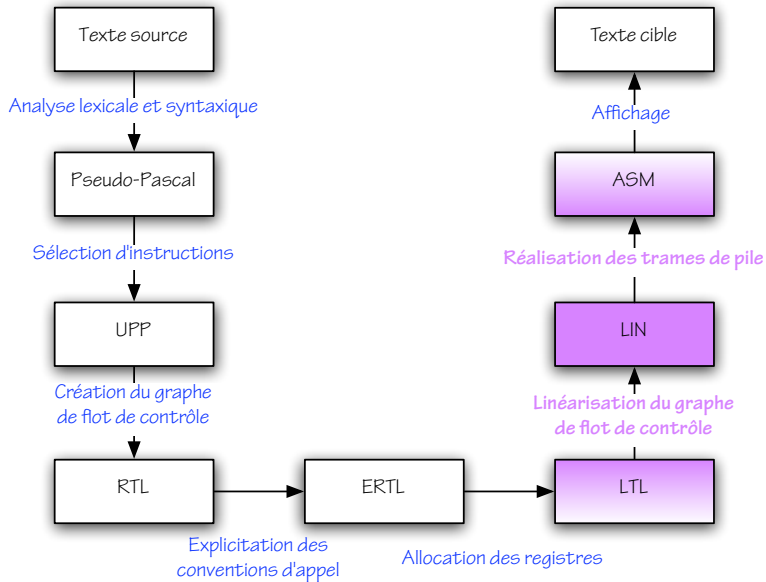
Sandrine Blazy
(d'après le cours de François Pottier)

ensiië - 2^e année

15 décembre 2008

**école nationale supérieure d'informatique
pour l'industrie et l'entreprise**

ensiië



1 De LTL à LIN

De LIN à ASM

De Pseudo-Pascal vers C ou Java

Ramassage de miettes

Exceptions

Location Transfer Language (LTL)

Voici une traduction de la fonction factorielle dans LTL :

```
procedure f(1)
var 8
entry f11
f11: newframe          → f10
f10: sets local(0), $ra → f9
f9 : j                → f8
f8 : sets local(4), $s0 → f7
f7 : move $s0, $a0     → f6
f6 : j                → f5
f5 : blez $s0          → f4, f3
f3 : addiu $a0, $s0, -1 → f2
f2 : j                → f20
f20: j                → f19
f19: call f           → f18
f18: j                → f1
f1 : mul $v0, $s0, $v0 → f0
f0 : j                → f17
f17: j                → f16
f16: gets $ra, local(0) → f15
f15: j                → f14
f14: gets $s0, local(4) → f13
f13: delframe         → f12
f12: jr $ra
f4 : li $v0, 1        → f0
```

Adieu, graphe de flot de contrôle

Dans LTL, les fonctions sont encore structurées sous forme de *graphe* de flot de contrôle.

Cela a été utile à plusieurs reprises. Par exemple, lors du passage de ERTL à LTL, des instructions de « spill » ont été *insérées*, et des instructions mortes ont été *supprimées* en les remplaçant par des IGoto.

Dans la suite, cette structure ne sera plus utile. On revient donc à une structure *linéaire*.

Élimination des IGoto

Dans LTL, l'instruction IGoto est *redondante*, puisque chaque instruction mentionne *explicitement* son ou ses successeurs.

On peut donc réaliser une transformation de LTL vers lui-même qui élimine tous les IGoto

Code Linéarisé (LIN)

Dans LIN,

- le graphe de flot de contrôle disparaît au profit d'une *suite linéaire* d'instructions ;
- le *successeur* de chaque instruction redevient implicite, sauf en cas de branchement ;
- les *labels* disparaissent, sauf pour les instructions cibles d'un branchement.

Code Linéarisé (LIN)

Voici une traduction de la fonction factorielle dans LIN :

```
procedure f(1)
var 8
f11:
newframe
sets local(0), $ra
sets local(4), $s0
move $s0, $a0
blez $s0, f4
addiu $a0, $s0, -1
call f
mul $v0, $s0, $v0
f16:
gets $ra, local(0)
gets $s0, local(4)
delframe
jr $ra
f4:
li $v0, 1
j f16
```


Linéarisation

La traduction de LTL vers LIN se fait par un simple *parcours* du graphe de flot de contrôle.

Lorsqu'on examine un sommet *pour la première fois*, on émet d'abord une *étiquette*, puis l'instruction associée à ce sommet, dont on examine ensuite les successeurs, en commençant par celui à qui le contrôle est transféré *implicitement*.

Lorsqu'on ré-examine un sommet *déjà rencontré*, on émet une instruction de *saut* inconditionnel vers l'étiquette correspondante.

On supprime a posteriori les étiquettes superflues.

Variations et critères de qualité

On peut vérifier que cet algorithme ne produit jamais de *saut* (conditionnel ou inconditionnel) *vers un saut* inconditionnel. (Pourquoi?)

Différents *ordres de parcours* des sommets donnent lieu à différentes linéarisations. *Inverser la condition* d'un saut conditionnel offre également une certaine latitude.

Certaines linéarisations peuvent être considérées comme *préférables* si elles utilisent le saut **j** en des points moins critiques...

Deux linéarisations d'une même boucle

La première exécute **j** à chaque itération, la seconde non.

```
...
début:
test:
(test)
bgtz $t1, fin
corps:
(corps)
j test
fin:
...

...
début:
j test
corps:
(corps)
test:
(test)
bgez $t1, corps
fin:
...
```

De LTL à LIN

2 De LIN à ASM

De Pseudo-Pascal vers C ou Java

Ramassage de miettes

Exceptions

Assembleur (ASM)

Dans ASM,

- la gestion des trames de pile se fait par *incréméntation* et *décréméntation* explicite du registre *sp* ;
- l'accès à la pile se fait à l'aide d'un *décalage fixe* vis-à-vis de *sp* ;
- la notion de procédure en tant qu'entité indépendante disparaît.

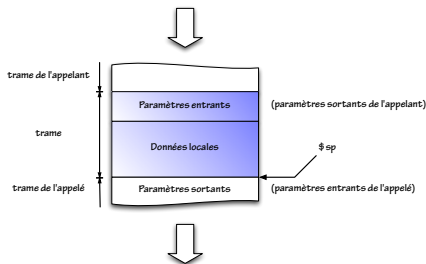
ASM est un *fragment* du langage assembleur MIPS et peut être aisément *affiché* sous forme textuelle, lisible par *spim*.

Assembleur (ASM)

Voici une traduction de la fonction factorielle dans ASM :

```
f11:                                f16:
addiu $sp, $sp, -8                lw    $ra, 4($sp)
sw    $ra, 4($sp)                 lw    $s0, 0($sp)
sw    $s0, 0($sp)                 addiu $sp, $sp, 8
move  $s0, $a0                    jr    $ra
blez  $s0, f4                      f4:
addiu $a0, $s0, -1                li    $v0, 1
jal   f11                          j     f16
mul   $v0, $s0, $v0
```

Organisation des trames de pile



La taille des deux régions qui forment la trame est enfin *connue*.

Un décalage *relatif* à l'une des trois régions « incoming », « local » ou « outgoing » peut donc être traduit en un simple décalage *vis-à-vis de \$sp*.

Les instructions **newframe** et **delframe** peuvent également être traduites en *décrémentations* et *incrémentations* de *\$sp*.

De LTL à LIN

De LIN à ASM

3 De Pseudo-Pascal vers C ou Java

Ramassage de miettes

Exceptions

Vers un compilateur C réel

Que nous manque-t-il? Relativement peu, en fait :

- un *préprocesseur* – qui peut être vu comme un outil séparé ;
- des types *entiers* et *flottants* de *diverses tailles*, avec *conversions* implicites ou explicites et *surcharge* des opérateurs arithmétiques ;
- des types *struct*, avec *allocation* dans les registres ou en pile, et *transmission* en tant qu'argument ou résultat ;
- l'opérateur **&**, avec répercussions sur *l'allocation* dans les registres ou en pile ;
- les constructions **for**, **do**, **switch**, **goto**, **break**, **continue** ;
- les *pointeurs de fonction* et les fonctions à *arité variable*.

Vers un compilateur Java

Que nous manque-t-il? De nombreux traits, parmi lesquels :

- un glaneur de cellules ou *GC* – cf. ce cours ;
- des *exceptions* – cf. ce cours ;
- des *classes*, *interfaces* et *objets* ;
- des *processus légers* (« *threads* ») ;
- un vérificateur de types *polymorphe*, pour Java 1.5 ;
- la *surcharge* des fonctions ou méthodes définies par l'utilisateur ;
- l'insertion de *tests* lors des accès aux *objets* et *tableaux* ;
- la production de « bytecode » exécutable par la *machine virtuelle Java* (JVM) ;
- ...

De LTL à LIN

De LIN à ASM

De Pseudo-Pascal vers C ou Java

4 Ramassage de miettes

Exceptions

Gestion manuelle ou automatique ?

Les objets alloués dans le tas – par *new array* en Pseudo-Pascal, *malloc* en C, *new* en Java – doivent-ils être désalloués *manuellement* ou *automatiquement* ?

Des dangers de la gestion manuelle

Par défaut, C exige une désallocation explicite à l'aide de la fonction *free*. Mais ce mécanisme *n'est pas sûr*. Il permet de :

- désallouer un objet *encore vivant* ;
- désallouer *deux fois* un objet.

Ces deux erreurs ne sont souvent *pas détectées immédiatement* et provoquent des comportements anormaux difficiles à analyser.

Des dangers de la gestion manuelle

Une gestion manuelle correcte doit obéir à des règles strictes qui devraient être clairement *comprises* et *documentées*. C'est rarement le cas.

On pourrait utiliser un *système de types*, c'est-à-dire faire appel au compilateur, pour imposer une discipline de gestion manuelle correcte.

Mais la conception de tels systèmes de types reste aujourd'hui un *problème largement ouvert* – ce qui reflète la difficulté de la question.

Des dangers de la gestion manuelle

De plus, une gestion manuelle souffre de deux inconvénients :

- *oublier* de désallouer un objet provoque une *fuite de mémoire* (« space leak ») difficile à détecter et à analyser ;
- devoir désallouer explicitement chaque objet a un *coût* que certains mécanismes de désallocation implicite permettent d'éviter...

Ramassage de miettes : principe

On considère le tas comme un *graphe* où les objets forment les sommets et les pointeurs forment les arêtes.

Les objets auxquels, à un instant donné, le programme est susceptible d'accéder *directement* – parce qu'il possède leur adresse – forment les *racines*.

Alors, tout objet *inaccessible* depuis une racine à travers le graphe est certainement *mort* (« garbage ») et peut être *désalloué*.

Ramassage de miettes : principe

De temps en temps – typiquement, lorsque le tas est plein – on examine le graphe pour déterminer quels objets sont accessibles, puis on désalloue les autres.

Ce processus s'appelle *ramassage de miettes*, « *garbage collection* » ou *glanage de cellules* (GC).

Ramassage de miettes : approximation

Il est possible que certains objets *accessibles* soient en fait d'ores et déjà et pour toujours *inutiles*. Ils ne seront pas désalloués, alors qu'ils pourraient l'être de façon sûre.

Le ramassage de miettes repose donc sur une *approximation* et peut être sujet à des *fuites de mémoire*. On doit éviter que des objets inutilisés ne restent pas accessibles par mégarde.

On peut vouloir obtenir une information plus fine par *analyse* du programme. Mais déterminer si un objet est vivant ou non est *indécidable*, donc une approximation restera nécessaire.

Un non-algorithme de ramassage de miettes

Le *comptage de références* (1960) consiste à stocker dans chaque objet un compteur du nombre actuel de pointeurs vers cet objet. Lorsque ce compteur devient nul, l'objet est désalloué.

Cette méthode *populaire* et souvent *réinventée* est en fait *inutilisable* :

- les sommets de degré entrant nul sont certes inaccessibles, mais *la réciproque est fausse*, d'où impossibilité de désallouer les structures cycliques ;
- la mise à jour du compteur à chaque fois qu'un pointeur est créé ou supprimé a un *coût prohibitif*.

Ramassage « mark & sweep »

L'idée (1960) est d'effectuer :

- d'abord un *parcours* du graphe pour déterminer et *marquer* les sommets accessibles ;
- puis un *balayage* linéaire du tas pour *désallouer* les objets non marqués et effacer les marques.

« mark & sweep » : pseudo-code

Voici la phase de *parcours*, ici parcours en profondeur d'abord :

```
procédure VISITER( $x$ )  
  si  $x$  n'est pas marqué  
  alors { marquer  $x$   
          pour chaque champ  $i$  de  $x$  faire VISITER( $x_i$ )  
        }  
  
procédure PARCOURIR()  
  pour chaque racine  $x$  faire VISITER( $x$ )
```

Quelles *difficultés* d'implémentation peut-on rencontrer ?

« mark & sweep » : pseudo-code

Voici la phase de *balayage* :

```
procédure BALAYER()  
  pour chaque objet  $x$   
    faire {  
      si  $x$  est marqué  
      alors effacer la marque  
      sinon ajouter  $x$  à une liste des objets disponibles
```

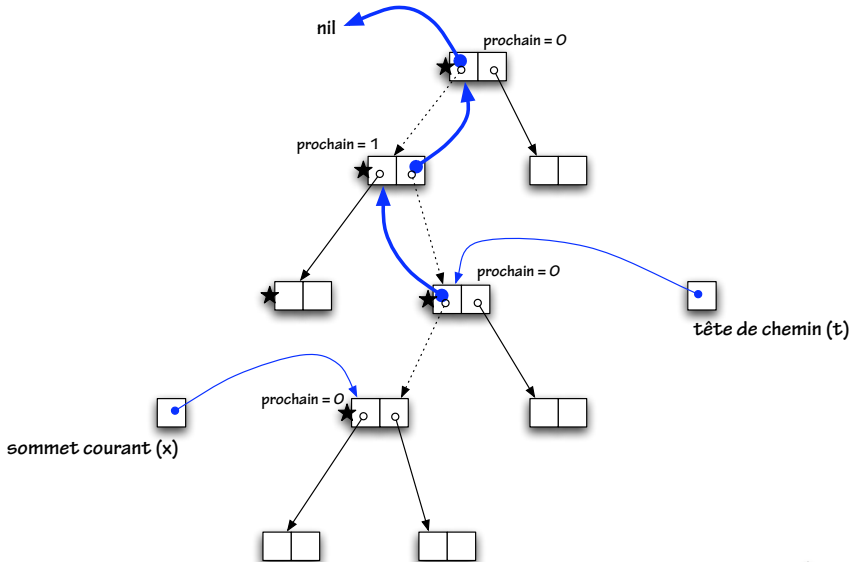
À quel *problème* peut-on être confronté ?

Parcours en profondeur sans pile externe

Pendant un parcours en profondeur d'abord, la pile reflète à chaque instant le *chemin parcouru* pour atteindre le sommet courant et permet ainsi de revenir en arrière.

L'idée, attribuée à Schorr et Waite, est de matérialiser ce *fil d'Ariane* non pas par une pile externe mais en *modifiant au vol le graphe* lui-même.

En gros, il suffit *d'inverser les arêtes* le long du chemin parcouru pour pouvoir revenir en arrière.



★ = marque

Parcours en profondeur à la Schorr–Waite

À chaque instant,

- x est le sommet en cours d'examen ;
- t est le sommet précédant x le long du chemin parcouru depuis la racine ;
- pour tout sommet z du chemin jusqu'à t , $i = z.\text{prochain}$ est *l'indice de l'arête suivie* à la descente, et le champ z_i contient l'adresse du *sommet précédant* z le long de ce chemin ;
- $x.\text{prochain}$ est l'indice de la prochaine arête à examiner.

Voir « *A case study of C source code verification : the Schorr–Waite algorithm* » (Hubert & Marché, 2005) pour un *invariant* complet.

Parcours en profondeur à la Schorr–Waite

procédure VISITER(x)

si x n'est pas marqué

$t \leftarrow \text{nil}$
marquer x
 $x.\text{prochain} \leftarrow 0$

alors

répéter

$i \leftarrow x.\text{prochain}$

si x admet un champ d'indice i

alors $\left\{ \begin{array}{l} \text{si } x_i \text{ n'est pas marqué} \end{array} \right.$

$\left\{ \begin{array}{l} x_i, t, x \leftarrow t, x, x_i \quad (\text{push}) \\ \text{marquer } x \end{array} \right.$

$\left\{ \begin{array}{l} x.\text{prochain} \leftarrow 0 \end{array} \right.$

sinon $x.\text{prochain} \leftarrow i + 1$

sinon $\left\{ \begin{array}{l} \text{si } t = \text{nil} \text{ alors terminé} \end{array} \right.$

$i \leftarrow t.\text{prochain}$

$x, t, t_i \leftarrow t, t_i, x \quad (\text{pop})$

$x.\text{prochain} \leftarrow i + 1$

« mark & sweep » : conclusion

Le coût du marquage est proportionnel à la taille de la partie accessible du tas, mais le *coût* du balayage est *proportionnel à la taille du tas* tout entier.

La *fragmentation* du tas peut provoquer des problèmes significatifs.

Ramassage « stop & copy »

L'idée (1970) est de *recopier* la partie accessible du graphe dans une *nouvelle* zone de mémoire.

Les objets accessibles une fois copiés seront *contigus*, d'où absence totale de fragmentation.

De plus, le *coût* de la copie est *proportionnel à la taille de la partie accessible* du graphe. À la limite, ce coût est *nul* si aucun objet n'est accessible !

Deux zones de mémoire

La zone où se situe initialement le tas est appelée « *from-space* ». La zone vers laquelle il est recopié est appelée « *to-space* ».

Une fois la copie terminée, les rôles des deux zones sont *intervertis* pour la prochaine phase de ramassage. De ce fait, 50% de la mémoire disponible est *inutilisée* en permanence.

Ce chiffre peut sembler élevé, mais on peut le diminuer en combinant la technique de « stop & copy » avec d'autres techniques, comme l'emploi de générations.

« Forwarding »

Lorsqu'un objet est copié, *son adresse change*. Il faut donc *faire suivre* (« forward ») tous les pointeurs vers lui.

Pour cela, il faut *mémoriser la correspondance* entre l'ancienne et la nouvelle adresse de l'objet. Mais où ?

Une fois l'objet situé dans le « from-space » copié vers le « to-space », *son contenu peut être écrasé*. On peut donc en réutiliser le premier champ (par exemple) pour stocker l'adresse de la copie.

Ce premier champ est alors appelé « *forwarding pointer* ».

« Forwarding »

Cette fonction *convertit* un « from-pointer » en un « to-pointer », en *copiant* l'objet au vol si nécessaire :

```
fonction FAIRESUIVRE( $x$ )  
  si  $x$  n'a pas été copié alors  $x_0 \leftarrow$  COPIER( $x$ )  
  renvoyer  $x_0$ 
```

La fonction Copier maintient un pointeur *next* vers la partie non encore utilisée du « to-space ». Elle *alloue* donc de l'espace dans le « to-space » en incrémentant ce pointeur.

Comment déterminer si x a déjà été copié ou non ?

« Forwarding » et « scanning »

Initialement, chaque objet, situé dans le « from-space », contient des pointeurs vers le « from-space ».

Lorsqu'un objet est copié, sa copie dans le « to-space » contient *toujours* des pointeurs vers le « from-space ».

Il faut donc examiner (« scan ») ces pointeurs les uns après les autres et les *faire suivre*.

Ce faisant, on découvrira éventuellement de nouveaux objets, qui seront copiés au vol ; il faudra donc examiner à leur tour ces nouvelles copies, et ainsi de suite.

« Scanning »

Un pointeur *scan* pointe vers le prochain objet à examiner dans le « to-space ».

Comme *next*, il va croissant. Tant qu'il reste des objets à examiner, il est strictement inférieur à *next*.

Lorsqu'il atteint *next*, il ne reste plus aucune copie à examiner. Donc, tous les pointeurs ont été redirigés, et tout est terminé.

« Scanning » : pseudo-code

L'algorithme complet s'écrit alors :

```
fonction RAMASSAGE()  
  scan ← next ← début du « to-space »  
  pour chaque racine x  
    faire x ← FAIRESUIVRE(x)  
  tant que scan < next  
    faire { soit x l'objet vers lequel pointe scan  
           pour chaque champ i de x  
             faire xi ← FAIRESUIVRE(xi)  
           incrémenter scan
```

De *quel type* de parcours de graphe s'agit-il ?

Interface entre compilateur et GC

Le compilateur doit :

- permettre d'identifier le *nombre* et la *nature* des champs de chaque objet ;
- permettre d'identifier les *racines*, qui se trouvent dans les *registres* et dans (toute!) la *pile* ;
- permettre le *marquage* des objets ;
- optimiser le code d'allocation par *expansion en ligne* (« inlining »), *fusion* des allocations effectuées au sein d'un même bloc de base, etc.

Ramassage à générations

Pour de nombreux programmes, les objets créés *le plus récemment* ont l'espérance de vie *la plus courte*. Inversement, les objets les plus anciens ont des chances de survivre encore longtemps.

Il est donc rentable de se concentrer sur les objets *jeunes*.

On réalise cela en divisant le tas en plusieurs zones ou *générations*. La génération G_{i+1} contient des objets plus anciens que la génération G_i . Le nombre de générations est au choix.

Ramassage à l'intérieur d'une génération

Pour effectuer un ramassage dans G_0 seule, par « mark & sweep » ou par « stop & copy », il faut connaître toutes les racines, *y compris les pointeurs de G_i , pour $i > 0$, vers G_0 .*

En pratique, ces pointeurs *d'un objet ancien vers un objet jeune* sont *peu fréquents*. Dans le cas d'un langage purement fonctionnel, ils sont en fait totalement inexistants !

Mais comment les déceler ? Il faut une forme de « *write barrier* : » un test doit être effectué à chaque fois que l'on *modifie* un champ appartenant à un objet de G_i . Voir Appel pour plus de détails.

Ramassage incrémental et concurrent

Pour certaines applications, il n'est *pas tolérable de stopper* le programme (le « *mutateur* ») pendant le temps nécessaire à un ramassage complet.

On souhaite alors que le « *collecteur* » travaille :

- en stoppant le programme, comme précédemment, mais par tranches incomplètes – *ramassage incrémental* ; ou bien,
- sans même stopper le programme, en tâche de fond – *ramassage concurrent*.

Ramassage incrémental et concurrent

Les algorithmes « mark & sweep » et « stop & copy » parcourent le graphe en distinguant des sommets *noirs* (visités et dont les fils ont été visités), *gris* (visités, mais dont les fils n'ont pas tous été visités), et *blancs* (non visités). Tout objet gris est *connu* du « collecteur » et en attente d'examen. De plus,

- *aucun objet noir n'a de fils blanc.*

La difficulté du ramassage incrémental ou concurrent consiste à *empêcher le « mutateur » de briser cet invariant* lorsqu'il crée un objet ou modifie un objet existant.

On réalise cela à l'aide de « write barriers » ou « read barriers ». Voir Appel pour plus de détails.

De LTL à LIN

De LIN à ASM

De Pseudo-Pascal vers C ou Java

Ramassage de miettes

5 Exceptions

Exceptions

En Java ou Objective Caml, *lancer une exception* transfère le contrôle directement au *gestionnaire d'exceptions* le plus récemment installé, en mettant fin à tous les appels de fonction commencés depuis.

Une *portion de pile* de taille arbitraire est donc supprimée en temps constant.

Voyons comment cela est réalisé en Objective Caml...

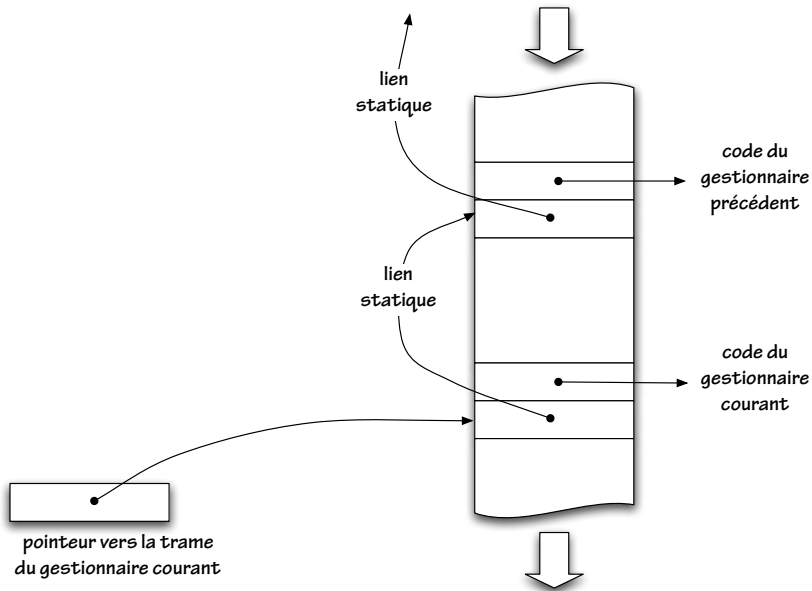
Les exceptions en Objective Caml

La construction **try/with** installe un gestionnaire d'exceptions. Une *trame* de deux mots est poussée sur la pile. Elle contient :

- un *pointeur vers le code* du gestionnaire ;
- un *lien statique*, c'est-à-dire un pointeur vers la trame du gestionnaire précédent.

Une *variable globale h* contient l'adresse de la trame du gestionnaire courant.

Les liens statiques réalisent une *liste chaînée* des trames associées aux gestionnaires installés, liste dont la tête est donnée par h .



Installation d'un gestionnaire

L'installation d'un nouveau gestionnaire consiste donc à installer sur la pile une trame pour ce gestionnaire et à mettre à jour *h*.

Si aucune exception n'est lancée pendant la durée de vie de ce gestionnaire, il est *désinstallé* par une manœuvre inverse.

Lancement d'une exception

Lorsqu'une exception est lancée,

- les registres « callee-save » sont restaurés (comment ?) ;
- l'exception, une valeur de type **exn**, est placée dans un registre dédié ;
- l'adresse de la trame du gestionnaire courant est lue dans h ;
- *la pile est coupée* juste au-dessus de cette trame ;
- le gestionnaire précédent est *réinstallé* ;
- le contrôle est transféré au gestionnaire courant.

Voilà !

Un aperçu de ce qu'est « *compiler* »...

...pourquoi ce n'est *pas simple*...

...et pourquoi un langage « *de haut niveau* » procure un réel avantage !