Langages, interprétation, compilation

Thibaut Balabonski @ Université Paris-Saclay
http://www.lri.fr/~blsk/CompilationLDD3/
v1.0, automne 2021
Troisième partie

Table des matières

6 Compilation **1**
6.1 *Back-end* 1
6.2 Représentation intermédiaire bas niveau 3
6.3 Raisonner sur les transformations 6
6.4 Traduction IMP vers LLIR 9
6.5 Architecture cible 16
6.6 Langage assembleur MIPS 19
6.7 Fonctions et conventions d'appel 25
6.8 Traduction LLIR vers MIPS 31
6.9 Tableaux 36
6.10 Gestion dynamique de la mémoire 39
6.11 Structures de données 43

7 Et ensuite **46**

6 Compilation

L'analyse préalable du programme source est terminée. Le vrai travail va pouvoir commencer.

6.1 *Back-end*

Les premières étapes de la compilation, sur lesquelles nous nous sommes concentrés jusque là, sont des étapes d'***analyse***, visant à extraire du sens du programme source reçu en entrée. Partant d'un programme source écrit dans un certain langage et donné sous la forme d'un fichier texte, nous avions donc :

1. une analyse *lexicale*, extrayant du texte source une séquence de lexèmes,
2. une analyse *grammaticale*, transformant cette séquence en un arbre de syntaxe, et
3. une analyse *sémantique*, vérifiant la cohérence du programme que l'on s'apprête à compiler.

Chacune de ces étapes est susceptible de s'interrompre et de renvoyer un message d'erreur à l'utilisateur, lorsque l'on détecte un problème dans le programme à compiler. Erreur lexicale par exemple si le texte source contient des symboles qui n'appartiennent pas au langage, erreur grammaticale par exemple si la forme du programme source est mauvaise, ou erreur sémantique si le programme source présente des incohérences de type. Toutes ces erreurs signalent un programme invalide qu'il est impossible (ou inutile) de compiler et demandent à l'utilisateur de corriger son programme avant de le soumettre à nouveau.

Phase de synthèse

Après cette série d'analyses, lorsque qu'elles passent toutes avec succès, vient une phase de ***synthèse***. Cette dernière phase vise à produire un programme cible équivalent au programme source, à nouveau sous la forme d'un fichier écrit dans un certain langage cible. Cette phase de synthèse, ou ***back-end***, du compilateur prend comme point de départ l'arbre de syntaxe abstraite du programme source, et enchaîne des phases d'optimisation et de traduction visant à produire un programme efficace écrit dans un langage cible. Cette étape n'est plus censée échouer : toute erreur dans le programme source susceptible de faire échouer la compilation doit avoir été remontée à l'utilisateur lors des phases d'analyse.

Le langage cible peut être par exemple :

— un autre langage de haut niveau, par exemple C ou javascript, et on pourra alors utiliser l'infrastructure de ce langage pour exécuter le programme obtenu,

— du ***bytecode***, c'est-à-dire du code bas niveau pour une machine virtuelle, par exemple les machines virtuelles de java, caml ou python,
— un langage ***assembleur***, pour exécution directe sur une architecture matérielle donnée, par exemple Intel x86, ARM ou MIPS (après assemblage et édition de liens).

La phase de synthèse comprend généralement plusieurs étapes, voire plusieurs dizaines d'étapes dans un compilateur industriel, allant progressivement du langage source vers le langage cible, en passant par plusieurs langages intermédiaires. On appelle ***représentations intermédiaires*** toutes les représentations à l'intérieur du compilateur de ces langages intermédiaires par lesquels va passer la traduction. Les représentations intermédiaires ont des formes variées, couvrant un spectre avec à une extrémité des arbres de syntaxe décrivant un programme structuré dans un langage encore d'assez haut niveau, et à l'autre extrémité des séquences d'instructions de bas niveau de type assembleur ou *bytecode*. Entre les deux, on peut trouver des représentations plus exotiques, par exemple basées sur des graphes.

Une petite optimisation

Une étape d'*optimisation* d'un compilateur travaille sur un programme exprimé dans l'une des représentations intermédiaires, et produit un programme équivalent mais « plus efficace ». L'optimisation elle-même est généralement la combinaison :

— d'une analyse, pour détecter ce qui peut être amélioré tout en respectant la sémantique du programme d'origine, et
— de la transformation elle-même, pour produire un programme résultat.

Le programme résultat peut être exprimé dans la même représentation intermédiaire que le programme d'origine.

Le compromis fondamental est de dépenser *une* fois du temps lors de la compilation pour en gagner *n* fois lors de l'exécution du programme. Les optimisations sont donc d'autant plus rentables qu'elles concernent des programmes ou des fragments de code destinés à être exécutés de nombreuses fois.

Une optimisation particulièrement simple consiste à simplifier les expressions constantes. En effet : ce qu'on peut calculer dès maintenant ne sera plus à calculer à l'exécution. En prenant la syntaxe abstraite du langage IMP par exemple, une expression `Add(Cst 4, Cst 7)` peut être simplifiée en l'expression `Cst 11`, et ce principe peut être appliqué récursivement pour simplifier autant que possible une expression complexe.

Un fragment d'une telle fonction de simplification pour IMP pourrait ainsi être :

```
let rec simplify = function
  | Add(e1, e2) ->
     let e1' = simplify e1 in
     let e2' = simplify e2 in
     begin match e1', e2' with
       | Cst n1, Cst n2 -> Cst (n1 + n2)
       | _, _ -> Add(e1', e2')
     end
```

Exercice : compléter cette fonction pour simplifier autant de choses que possible tout en préservant la sémantique du programme source.

Pour aller plus loin, on pourrait aussi imaginer propager les valeurs des variables lorsqu'elles sont connues. Ainsi dans un code comme

```
x = 1;
y = x+1;
```

on peut savoir à la deuxième ligne que la valeur de `x` est `1`, et en déduire une simplification de la deuxième affectation en `y = 2;`. Cependant, les simplifications de ce type peuvent demander des analyses beaucoup plus fines. Peut-on de même simplifier l'affectation à `y` dans un programme de la forme suivante ?

```
x = 1;
while (...) {
  y = x+1;
  x = 1/x;
}
```

Nous n'irons pas jusque là dans ce cours (rendez-vous l'année prochaine pour de vraies optimisations).

Fil conducteur

Dans ce cours, nous allons dans un premier temps prendre comme langage source le langage IMP, et développer un schéma simple de compilation vers le langage assembleur MIPS, en trois étapes.

1. Nous venons de voir une mini-étape d'optimisation, sur l'arbre de syntaxe IMP (première représentation)
2. Traduction vers le code d'une machine virtuelle de bas niveau (deuxième représentation).
3. Traduction du code obtenu à l'étape précédente vers du code assembleur MIPS (troisième représentation).

À la fin du chapitre nous ajouterons de nouvelles phases préalables, partant d'un langage source d'un peu plus haut niveau inspiré de C, qui sera traduit vers IMP.

6.2 Représentation intermédiaire bas niveau

Voici un exemple de code bas niveau, exprimé dans ce qui sera notre deuxième représentation intermédiaire pour notre compilateur IMP.

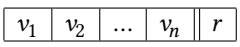
```
function fact {
  nb_params: 1
  nb_locals: 0
  start: fact_4

  fact_4:
    cst 2; push; get param[0]; lt; cjump fact_2, fact_3

  fact_3:
    cst -1; push; get param[0]; add; push; call fact;
    push; get param[0]; mul; return

  fact_2:
    cst 1; return
}
```

Nous allons bientôt détailler cette représentation, mais vous pouvez déjà déceler que l'ensemble de ce code exemple représente une fonction, dont le corps est séparé en trois blocs, chacun introduit par une étiquette de la forme `fact_*`. Chaque bloc est ensuite constitué d'une séquence d'instructions élémentaires, avec des opérations arithmétiques élémentaires (`cst`, `add`, `mul`), des accès à des variables (`get`), des appels de fonctions (`call`, `return`), des sauts (`cjump`), des manipulations d'une pile (`push`)...

Cette représentation intermédiaire, que nous appellerons LLIR, peut être vue comme un code pour une ***machine virtuelle***, c'est-à-dire un ordinateur fictif, dont l'architecture serait centrée autour de deux éléments :

— un registre de travail, contenant la valeur de l'expression courante, et
— une pile, stockant les valeurs intermédiaires « en attente ».

On pourra représenter un état de la machine virtuelle par un tableau



où les v_i sont les valeurs intermédiaires stockées sur la pile, avec v_1 la plus ancienne et v_n la plus récente, et où r est la valeur contenue dans la registre.

Un programme LLIR est constitué d'un ensemble de déclarations de variables globales, et d'un ensemble de définitions de fonctions.

Arithmétique

Cette architecture est facilement exploitable pour réaliser des calculs arithmétiques. On peut par exemple produire la valeur de l'expression `(1+2)+(4+8)` avec la séquence d'actions suivante.

1. Partir de l'état « vide ». 

2. Calculer la valeur de `1+2`.

(a) Placer la valeur `1` dans le registre. | 1 |
(b) Sauvegarder le registre sur la pile. | 1 | 1 |
(c) Placer la valeur `2` dans le registre. | 1 | 2 |
(d) Ajouter au registre la dernière valeur de la pile. | 3 |

3. Sauvegarder la valeur de `1+2` sur la pile. | 3 | 3 |
4. Calculer la valeur de `4+8`.
   (a) Placer la valeur `4` dans le registre. | 3 | 4 |
   (b) Sauvegarder le registre sur la pile. | 3 | 4 | 4 |
   (c) Placer la valeur `8` dans le registre. | 3 | 4 | 8 |
   (d) Ajouter au registre la dernière valeur de la pile. | 3 | 12 |
5. Ajouter au registre la dernière valeur de la pile. | 15 |

Notre représentation intermédiaire fournit donc des instructions élémentaires pour chacune des opérations effectuées dans cet exemple.

— L'instruction `cst` n place dans le registre une valeur numérique n.

| p | r | → | p | n |

— L'instruction `push` sauvegare sur la pile la valeur courante du registre.

| p | r | → | p | r | r |

— Les instructions `add`, `mul`, `lt`, ... appliquent une opération aux valeurs contenues dans le registre et au sommet de la pile.

| p | v | r | → | p | $r \star v$ |

Notez que ces instructions consomment la valeur présente au sommet de la pile, et que leur comportement n'est pas défini dans le cas où la pile serait vide.

Dans ce cadre, *compiler* une expression c'est produire un fragment de code LLIR qui calcule la valeur de cette expression et laisse le résultat dans le registre. Ainsi on compile l'expression e_1 `+` e_2 en produisant l'enchaînement suivant :

1. commencer avec le code évaluant la sous-expression e_1,
2. ajouter une instruction `push` pour sauvergarder le résultat sur la pile,
3. poursuivre avec le code évaluant la sous-expression e_2,
4. conclure avec une instruction `add`.

Remarquez que l'instruction `add` est placée un peu après une instruction `push`, et s'exécutera donc bien avec une pile non vide. Ou du moins, c'est le cas si l'on suppose que la séquence d'instructions relative à e_2 ne fait pas n'importe quoi. Nous en reparlerons dans la section suivante.

Variables

On a dans LLIR trois catégories de variables.

— Les variables globales, désignées par un identifiant. Par exemple : `x`.
— Les variables locales, qui sont numérotées, avec la notation `local[`k`]`.
— Dans le cadre d'un appel de fonction, les paramètres de cet appel, qui sont numérotés également, avec la forme `param[`k`]`.

Ces trois catégories de variables sont manipulées de la même manière dans le code, à l'aide de deux instructions.

— L'instruction d'accès en lecture `get` v place la valeur de la variable v dans le registre.
— L'instruction d'accès en écriture `set` v définit la nouvelle valeur de la variable v avec la valeur courante du registre.

Dans le langage IMP, ces trois sortes de variables existent mais elles sont toutes désignées exactement de la même façon : par des identifiants. La seule distinction entre les trois versions est la manière dont elles sont déclarées :

— les variables globales sont déclarées en tête du fichier,
— les variables locales sont déclarées en tête du corps d'une fonction,
— les paramètres d'une fonction sont déclarés dans la signature de la fonction elle-même.

Pour permettre la traduction des identifiants IMP vers des variables LLIR on se base sur une ***table des symboles***, c'est-à-dire une table associant des informations à chaque

identifiant du programme source IMP. Les informations contenues dans une table des symboles peuvent être de différentes natures. Ici on se contente de renseigner la manière dont chaque variable est déclarée : globale, locale ou comme paramètre de fonction. Dans les deux derniers cas, on ajoute un numéro localement unique, en commençant par exemple à 1 la première variable locale et pour le premier paramètre d'une fonction. Cette table peut être déduite simplement des informations contenues dans l'AST.

La compilation des accès aux variables peut alors suivre les schémas suivants.

— On traduit une expression IMP `x` par une simple instruction `get` v, où v est la désignation de la variable `x` au format LLIR, obtenue en consultant la table des symboles pour `x`.
— On traduit une instruction IMP `x = e;` par l'enchaînement suivant :
  1. d'abord le code évaluant `e`,
  2. puis l'instruction `set` v, avec v la notation LLIR associée à l'identifiant `x`.

Contrôle

Le code d'une fonction LLIR est donné par un ensemble de ***blocs***, qui forment l'unité la plus fine d'organisation du code. Un bloc est une séquence d'instructions qui :

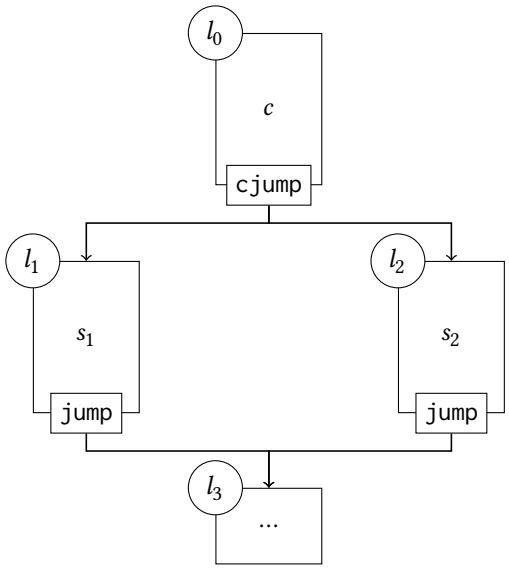
— a un *nom unique*, donné par une étiquette,
— est *linéaire*, c'est-à-dire qui ne contient aucun branchement,
— *termine par un saut* vers le code à exécuter ensuite, avec éventuellement plusieurs suites possibles, ou par une instruction `return`.

On a deux instructions de saut possibles pour conclure un bloc, en plus de `return`.

— Le saut inconditionnel `jump` l désigne comme bloc suivant le bloc d'étiquette l.
— Le saut conditionnel `cjump` l_1 l_2 désigne comme bloc suivant l'un des blocs d'étiquette l_1 ou l_2. Le choix ira vers l_1 si la valeur courante du registre est non nulle, ou vers l_2 sinon.

Les structures de contrôle du langage IMP sont traduites par des combinaisons de blocs LLIR. On peut les représenter à l'aide de graphes dont les nœuds représentent des blocs d'instructions et les arcs les instructions de saut.

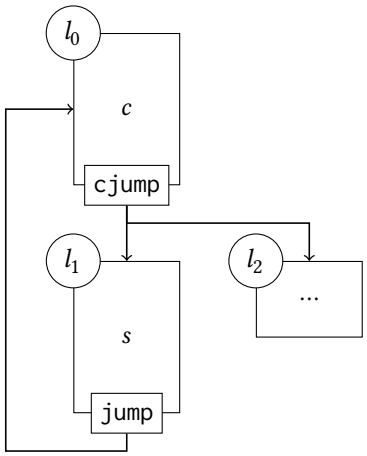
— Une instruction de branchement conditionnel `if (`c`) { `s_1` } else { `s_2` }` introduit un graphe comportant un branchement au niveau du test, et un point de jonction auquel on saute après l'exécution de l'une ou l'autre des séquences s_1 ou s_2.



Selon le contenu des séquences s_1 et s_2, les parties correspondantes peuvent elles-mêmes être composées de plusieurs blocs. Le bloc de jonction peut être directement le premier bloc de l'instruction suivant le branchement.

— Une boucle `while (`c`) { `s` }` donne un graphe contenant un cycle, où après l'exécution de la séquence s on saute à nouveau au bloc effectuant le test de la

condition.



Le bloc de sortie peut être directement le premier bloc de l'instruction suivant la boucle.

Fonctions

Une fonction LLIR est définie par :

— son nom,
— son nombre de paramètres,
— son nombre de variables locales,
— un ensemble de blocs de code,
— l'identification du bloc de départ.

Notez que les blocs n'ont pas d'ordre particulier : l'identification du bloc de départ suffit à commencer l'exécution, puis les sauts à la fin de chaque bloc donneront la succession à utiliser. Les identifiants des paramètres et des variables locales ne sont pas non plus utiles : on accède à ces variables uniquement par leur numéro, et connaître leur nombre suffira donc.

On a deux instructions pour la gestion des appels de fonction.

— L'instruction `return` déjà mentionnée permet d'arrêter l'exécution de la fonction. La valeur renvoyée par la fonction est celle qui se trouve dans le registre.
— L'instruction `call` f déclenche un appel à la fonction de nom f. Les paramètres sont pris sur la pile et en sont retirés. Le résultat renvoyé est placé dans le registre.

Chaque appel de fonction s'exécute en isolation des autres, avec sa propre pile, son propre tableau de paramètres et son propre tableau de variables locales. Seules les variables globales sont partagées.

Un appel de fonction `f(`e_1`, ..., `e_n`)` du langage IMP est compilé en l'enchaînement suivant :

— évaluer e_n et empiler le résultat,
— ...
— évaluer e_1 et empiler le résultat,
— conclure avec `call f` pour réaliser l'appel lui-même.

Avec tout ceci en main, vous pouvez relire et interpréter le code de la fonction factorielle en LLIR vu plus haut.

6.3 Raisonner sur les transformations

On a décrit un compilateur comme un programme qui :

— prend en entrée un programme source P_1, et qui
— produit un programme cible P_2 *équivalent.*

Par « équivalent » on entend informellement « qui a le même comportement en toutes circonstances », ou autrement dit « qui a le même comportement pour chacune des entrées possibles ». Plus précisément, voici ce qu'on attend de l'exécution du programme P_2 sur des entrées e_1, ..., e_n en fonction du comportement de P_1 sur ces mêmes entrées.

— Si P_1 est défini et s'exécute sans erreur, alors P_2 doit produire le même résultat et les mêmes effets (et ne pas faire d'erreur). Petite subtilité : si P_1 n'est

pas déterministe, alors on peut demander à P_2 de produire l'un des ensembles résultat/effets possibles pour P_1.
- Si P_1 produit une erreur, alors P_2 doit produire une erreur.
- Si le comportement de P_1 est indéfini, alors P_2 peut avoir un comportement arbitraire (il peut produire un résultat, ou produire une erreur, ou être lui aussi indéfini).

Ce contrat est relatif aux sémantiques du langage source et du langage cible.

Cette notion de ***correction*** d'un compilateur peut s'appliquer à toute transformation de programme, et en particulier à chacune des étapes d'optimisation et de traduction de la phase de synthèse. Autrement dit, on peut justifier qu'un compilateur donné est correct en justifiant que chacune des transformations qu'il effectue préserve bien le comportement des programmes.

Pour énoncer la correction d'une transformation de programme, il faut au préalable :
- une sémantique du langage source,
- une sémantique du langage cible (si différent du langage source),
- une fonction de traduction.

Ainsi, démontrer la correction d'un compilateur complet en analysant les étapes de transformation l'une après l'autre demande d'avoir, en plus d'une sémantique pour le langage source : une sémantique du langage cible, et une sémantique de chaque représentation intermédiaire !

Cas : optimisation des expressions constantes

On a mentionné une optimisation de simplification des expressions constantes pour le langage IMP. Il s'agit d'une transformation dont le langage source et le langage cible sont tous les deux IMP. On justifie donc sa correction en fonction d'une sémantique de IMP.

Supposons par exemple les règles simples suivantes pour un fragment des expressions du langage. On utilise les notations caml de la syntaxe abstraite pour éviter les confusions entre syntaxe et sémantique.

$$\frac{}{\texttt{Cst}\ n, \rho \implies n} \qquad \frac{}{\texttt{Var}\ x, \rho \implies \rho(x)} \qquad \frac{e_1, \rho \implies n_1 \qquad e_2, \rho \implies n_2}{\texttt{Add}(e_1, e_2), \rho \implies n_1 + n_2}$$

La simplification des expressions constantes réalise une fonction f satisfaisant les équations suivantes.

$$\begin{array}{rcl} f(\texttt{Cst}\ n) & = & \texttt{Cst}\ n \\ f(\texttt{Var}\ x) & = & \texttt{Var}\ x \\ f(\texttt{Add}(e_1, e_2)) & = & \begin{cases} \texttt{Cst}\ (n_1 + n_2) & \text{si } f(e_1) = n_1 \text{ et } f(e_2) = n_2 \\ \texttt{Add}(f(e_1), f(e_2)) & \text{sinon} \end{cases} \end{array}$$

On se donne l'énoncé de correction suivant :

$$\text{Pour tous } e, \rho, \text{ si } e, \rho \implies n \text{ alors } f(e), \rho \implies n.$$

On le démontre par récurrence sur la dérivation de $e, \rho \implies n$.
- Cas $n, \rho \implies n$ et $x, \rho \implies \rho(x)$. Dans ces cas la fonction f est l'identité, et la conclusion est immédiate.
- Cas $\texttt{Add}(e_1, e_2) \implies n_1 + n_2$ avec $e_1, \rho \implies n_1$ et $e_2, \rho \implies n_2$. Les deux prémisses nous donnent les hypothèses de récurrence $f(e_1), \rho \implies n_1$ et $f(e_2), \rho \implies n_2$.
  La définition de $f(\texttt{Add}(e_1, e_2))$ dépend des formes de $f(e_1)$ et de $f(e_2)$.
  - Si $f(e_1) = n_1'$ et $f(e_2) = n_2'$, alors $f(\texttt{Add}(e_1, e_2)) = \texttt{Cst}(n_1' + n_2')$. Dans ce cas, l'hypothèse de récurrence $f(e_1), \rho \implies n_1$ s'écrit en outre $n_1', \rho \implies n_1$, ce qui n'est possible qu'avec $n_1' = n_1$. De même on a $n_2' = n_2$, et donc $f(\texttt{Add}(e_1, e_2)) = \texttt{Cst}(n_1' + n_2') = \texttt{Cst}(n_1 + n_2)$.
  - Sinon $f(\texttt{Add}(e_1, e_2)) = \texttt{Add}(f(e_1), f(e_2))$, et à l'aide des hypothèse de récurrence on peut déduire

$$\frac{f(e_1), \rho \implies n_1 \qquad f(e_2), \rho \implies n_2}{\texttt{Add}(f(e_1), f(e_2)), \rho \implies n_1 + n_2}$$

Cas : traduction des expressions arithmétiques

Considérons maintenant la traduction des expressions arithmétiques du langage IMP en en séquences d'instructions LLIR. La fonction de traduction satisfait les équations suivantes.

$$
\begin{array}{rcl}
f(\texttt{Cst}\ n) & = & \texttt{cst}\ n \\
f(\texttt{Var}\ x) & = & \begin{cases} \texttt{get}\ x & \text{si } x \text{ variable globale} \\ \texttt{get local[}k\texttt{]} & \text{si } x \text{ est la } k\text{ème variable locale} \\ \texttt{get param[}k\texttt{]} & \text{si } x \text{ est le } k\text{ème paramètre} \end{cases} \\
f(\texttt{Add}(e_1, e_2)) & = & f(e_1);\ \texttt{push};\ f(e_2);\ \texttt{add}
\end{array}
$$

La correction de cette transformation exprime un lien entre l'évaluation de l'expression IMP selon la sémantique déjà connue, et l'exécution des instructions LLIR. La sémantique de ces dernières doit encore être formalisée !

Les instructions LLIR sont exécutées par une machine virtuelle utilisant un registre, une pile et une mémoire. Chaque instruction exécutée modifie certains de ces éléments, qui forment ensemble l'***état*** de la machine. Ainsi pour une séquence $i_1\,;\ i_2\,;\ \ldots i_n$ on a les évolutions successives

$$S_0 \xrightarrow{i_1} S_1 \xrightarrow{i_2} S_2 \ldots \xrightarrow{i_n} S_n$$

où chaque état S_k est décrit par un triplet

$$[\rho \mid r \mid \pi]$$

où

- ρ est un environnement, c'est-à-dire une représentation de la mémoire sous la forme d'une fonction associant des valeurs aux variables,
- r est la valeur courante du registre,
- π est une liste de valeurs représentant la pile.

On formalise la sémantique en définissant les transitions $S \xrightarrow{i} S'$ d'un état S à un état S' sous l'effet d'une instruction i.

état de départ	instruction	état d'arrivée
$[\rho \mid r \mid \pi]$	$\texttt{cst}\ n$	$[\rho \mid n \mid \pi]$
$[\rho \mid r \mid \pi]$	$\texttt{get}\ v$	$[\rho \mid \rho(v) \mid \pi]$
$[\rho \mid r \mid \pi]$	push	$[\rho \mid r \mid r::\pi]$
$[\rho \mid r \mid n::\pi]$	add	$[\rho \mid n + r \mid \pi]$

Note : par v on désigne ici l'une quelconque des formes x, $\texttt{local[}k\texttt{]}$ ou $\texttt{param[}k\texttt{]}$.

L'énoncé de correction de la traduction des expressions exprime un lien entre la relation d'évaluation $e, \rho \implies n$ et l'évolution de l'état de la machine virtuelle LLIR sous l'effet de l'exécution d'une séquence d'instructions.

Pour tous e, ρ, r et π, si $e, \rho \implies n$ alors $[\rho \mid r \mid \pi], f(e) \to^* [\rho \mid n \mid \pi]$.

Notez dans cet énoncé que le contenu initial du registre et de la pile n'ont aucune influence. La pile doit en revanche, à la fin de l'exécution, avoir retrouvé son état initial.

On démontre cet énoncé par récurrence sur la dérivation de $e, \rho \implies n$.

- Cas $n, \rho \implies n$. Alors $f(n) = \texttt{cst}\ n$ et la règle de transition correspondante donne directement le résultat souhaité : $[\rho \mid r \mid \pi], \texttt{cst}\ n \to [\rho \mid n \mid \pi]$.
- Cas $x, \rho \implies \rho(x)$ similaire.
- Cas $\texttt{Add}(e_1, e_2), \rho \implies n_1 + n_2$ avec $e_1, \rho \implies n_1$ et $e_2, \rho \implies n_2$. Les deux prémisses nous donnent les hypothèses de récurrence « $\forall r, \pi,\ [\sigma \mid r \mid \pi], f(e_1) \to^* [\sigma \mid n_1 \mid \pi]$ » et « $\forall r, \pi,\ [\sigma \mid r \mid \pi], f(e_2) \to^* [\sigma \mid n_2 \mid \pi]$ ».
  On rappelle que $f(\texttt{Add}(e_1, e_2)) = f(e_1);\ \texttt{push};\ f(e_2);\ \texttt{add}$, dont on déduit la séquence d'exécution suivante.

$$
\begin{array}{lcll}
[\sigma \mid r \mid \pi] & , f(e_1) \to^* & [\sigma \mid n_1 \mid \pi] & \text{par hyp. de récurrence} \\
[\sigma \mid n_1 \mid \pi] & , \texttt{push} \to & [\sigma \mid n_1 \mid n_1::\pi] & \\
[\sigma \mid n_1 \mid n_1::\pi], f(e_1) & \to^* & [\sigma \mid n_2 \mid n_1::\pi] & \text{par hyp. de récurrence} \\
[\sigma \mid n_2 \mid n_1::\pi], \texttt{add} & \to & [\sigma \mid n_1 + n_2 \mid \pi] &
\end{array}
$$

À la fin de la séquence, on a bien obtenu la valeur $n_1 + n_2$ dans le registre, et la pile a retrouvé son état d'origine π. Notez que la deuxième application de l'hypothèse de récurrence est faite dans un état où la pile a été étendue par rapport à son état initial. C'est possible grâce à la quantification universelle sur π dans l'hypothèse de récurrence.

6.4 Traduction IMP vers LLIR

Dans cette section, nous allons présenter le code complet d'une traduction de IMP vers LLIR.

Syntaxe abstraite IMP

Un programme IMP contient des variables globales et des définitions de fonctions.

```
type prog = {
  globals: string list;
  functions: function_def list;
}
```

Chaque définition de fonction comporte un nom, une liste de paramètres formels, une liste de variables locales et un code.

```
type function_def = {
  name: string;
  params: string list;
  locals: string list;
  code: seq;
}
```

Le code est une séquence d'instructions comportant notamment des opérations d'affectation et les instructions de contrôle classiques. On y ajoute une primitive `putchar` d'affichage d'un caractère, une instruction `return` pour la fin d'exécution des fonctions, et la possibilité d'utiliser une expression à la place d'une instruction.

```
type instr =
  | Set of string * expr (* x = e; *)
  | If of expr * seq * seq (* if (e) { s1 } else { s2 } *)
  | While of expr * seq  (* while (e) { s } *)
  | Putchar of expr      (* putchar(e); *)
  | Return of expr       (* return(e); *)
  | Expr of expr         (* e; *)
and seq = instr list     (* i1; i2; ...; iN *)
```

Les expressions sont formées avec quelques opérations arithmétiques déjà manipulées, des variables, et des appels de fonctions.

```
type expr =
  | Cst of int               (* 17 *)
  | Add of expr * expr       (* e1 + e2 *)
  | Mul of expr * expr       (* e1 * e2 *)
  | Lt of expr * expr        (* e1 < e2 *)
  | Get of string            (* x *)
  | Call of string * expr list (* f(e1, ..., eN) *)
```

Représentation intermédiaire LLIR

Un programme au format LLIR est composé, comme au format IMP, d'un ensemble de variables globales et d'un ensemble de fonctions. Les définitions de fonction en revanche évoluent. D'une part, les variables locales et les paramètres formels des fonctions sont identifiés en LLIR par leur numéro. Il n'est donc plus nécessaire de retenir leurs identifiants, leur nombre suffit. D'autre part, le code d'une fonction LLIR est donné par un ensemble de blocs d'instructions, chacun associé à une étiquette. Il n'y a plus ici de notion d'ordre entre les différents blocs. À la place, chaque bloc termine par une instruction de saut permettant d'identifier le bloc à exécuter ensuite. On peut voir ce code comme un *graphe* dont les sommets sont des blocs d'instructions et dont les arêtes correspondent aux successions possibles entre blocs. On représente cet ensemble par une table de hachage où les clés sont les étiquettes et les valeurs sont les blocs associés.

```
type function_def = {
  name: string;
  nb_params: int;
  nb_locals: int;
  code: (string, seq) Hashtbl.t;
  start: string;
}

type prog = {
  globals: string list;
  functions: function_def list;
}
```

On introduit un type commun pour représenter les différentes sortes de variables des programmes LLIR (variables globales, paramètres formels de fonctions, variables locales de fonctions).

```
type var =
  | Global of string
  | Param of int
  | Local of int
```

Les instructions enfin sont maintenant restreintes à des opérations élémentaires, agissant sur le registre, sur la pile ou sur la mémoire. On y retrouve quelques opérations élémentaires d'arithmétique,

```
type instr =
  | Cst of int (* r <- n *)
  | Push       (* push r *)
  | Add        (* r <- r + pop *)
  | Mul        (* r <- r * pop *)
  | Lt         (* r <- (pop < r)?1:0 *)
```

des instructions de lecture ou d'écriture des variables,

```
  | Get of var (* r <- var *)
  | Set of var (* var <- r *)
```

ou des instructions de saut et de saut conditionnel.

```
  | Jump of string         (* exec l *)
  | CJump of string * string (* exec r?l1:l2 *)
```

L'instruction d'appel de fonction prend les paramètres de la fonction sur la pile, et le résultat est renvoyé via le registre.

```
  | Call of string (* r <- f(pop, ..., pop) *)
  | Return        (* return r *)
```

On conserve une opération primitive d'affichage (prenant en paramètre le registre), ainsi qu'une notion de séquence.

```
  | Putchar    (* print r *)
type seq = instr list
```

Traduction de IMP vers LLIR

Pour traduire un programme IMP en programme LLIR, on traduit indépendamment chacune des fonctions à l'aide d'une fonction `tr_fdef` définie ci-après.

```
let tr_prog prog = {
  Llir.globals = Imp.(prog.globals);
  Llir.functions = List.map tr_fdef Imp.(prog.functions);
}
```

La principale tâche de la fonction caml traduisant les fonctions IMP en fonctions LLIR consiste à produire le graphe des blocs d'instructions LLIR. En l'occurrence, on produira au moins un bloc LLIR pour chaque instruction IMP. On définit pour cela

une table de hachage pour les blocs et deux fonctions auxiliaires : `new_name` produit de nouveaux noms de blocs (préfixés par le nom de la fonction) et `new_block` crée un nouveau bloc à partir d'une séquence d'instructions LLIR.

```
let tr_fdef fdef =
  let code = Hashtbl.create 32 in
  let new_name =
    let f = Imp.(fdef.name) in
    let cpt = ref 0 in
    fun () -> incr cpt; Printf.sprintf "%s_%i" f !cpt
  in
  let new_block b =
    let block_name = new_name() in
    Hashtbl.add code block_name b;
    block_name
  in

  ...
```

Notez que `new_block` génère à la volée un nom pour le nouveau bloc `b`, et renvoie ce nom. En outre, on aura besoin de connaître ce nom pour former la séquence d'instructions du bloc « précédent » du programme, puisque cette séquence « précédente » devra terminer par une instruction de saut vers le bloc `b`. Autrement dit, les blocs LLIR correspondant à une séquence IMP vont être créés à rebours, en commençant par la dernière instruction IMP et en revenant progressivement à la première. Ainsi, la fonction `tr_instr` traduisant une instruction IMP en un bloc LLIR prend en paramètre supplémentaire le nom du bloc suivant. De même, la fonction `tr_seq` de traduction d'une séquence IMP prend en paramètre supplémentaire le nom du bloc suivant, et renvoie le nom du premier bloc. Pour traduire une séquence `i::s` on traduit donc d'abord sa queue `s`, après quoi on récupère le nom `next'` du premier bloc de la queue, qui sera désigné comme successeur du bloc de tête.

```
  let rec tr_seq s next = match s with
    | i::s -> let next' = tr_seq s next in
              new_block (tr_instr i next')
```

Notez que la fonction `tr_seq`, ainsi que toutes les fonctions restant à venir dans ce traducteur, sont définies comme des fonctions internes de `tr_fdef`. Elles ont ainsi bien accès à la table de hachage `code` et à la fonction `new_block`.

On complète la fonction `tr_seq` en traitant à part le cas d'une séquence d'une unique instruction et le cas d'une séquence vide, pour remplacer ce dernier cas par une instruction anodine.

```
    | [i] -> new_block (tr_instr i next)
    | []  -> new_block (tr_instr Imp.(Expr(Cst 0)) next)
  in
```

En poussant cette logique à l'échelle de la fonction IMP complète, le premier bloc créé va être le bloc final de la fonction, puis on progressera petit à petit en revenant vers le début du texte de la fonction. Le code principal de `tr_fdef` crée donc d'abord un bloc final avant de traduire le reste du code.

```
  let end_name = new_block [Llir.Cst 0; Llir.Return] in
  let start = tr_seq Imp.(fdef.code) end_name in
```

Ici, le bloc final correspond à une instruction `return(0);` ajoutée à la fin du programme, qui sera exécutée si aucune instruction `return` n'est rencontrée plus tôt. Une fois passé cet appel à `tr_seq`, la table de hachage `code` contient tous les blocs de code LLIR traduisant le code IMP de la fonction, et `start` désigne le bloc de tête. Il ne reste plus qu'à renseigner les champs de la structure représentant la fonction LLIR.

```
  {
    Llir.name = Imp.(fdef.name);
    Llir.nb_params = Imp.(List.length fdef.params);
    Llir.nb_locals = Imp.(List.length fdef.locals);
    Llir.code = code;
```

```
  Llir.start = start;
}
```

On crée un nouveau bloc pour chaque instruction IMP traduite, voire plusieurs blocs pour les instructions impliquant des branchements. La fonction de traduction `tr_instr` prend en paramètre supplémentaire le nom du bloc suivant, et renvoie une séquence d'instructions LLIR destinée à former un nouveau bloc. La fonction `tr_instr` fait appel à une fonction `tr_expr` dédiée à la traduction des expressions. La fonction `tr_expr` prend en paramètre, outre une expression `e` à traduire, une liste `k` d'instructions LLIR à exécuter *après* l'évaluation de `e`. Le résultat d'un appel `tr_expr e k` est donc une liste d'instructions LLIR contenant d'abord les instructions évaluant `e`, puis la liste « de suite » `k`.

Ainsi par exemple, on traduit une instruction IMP `putchar(e);` précédant un bloc de nom `next` en faisant suivre les instructions d'évaluation de `e` de l'instruction LLIR `putchar` et de l'instruction de saut `jump next`.

```
let rec tr_instr i next = match i with
  | Imp.Putchar(e) ->
    tr_expr e [Llir.Putchar; Llir.Jump next]
```

On suit la même logique pour les instructions d'affectation, d'évaluation d'une expression seule, ou de retour.

```
  | Imp.Set(x, e) ->
    tr_expr e [Llir.Set (convert_var x); Llir.Jump next]
  | Imp.Expr e ->
    tr_expr e [Llir.Jump next]
  | Imp.Return e ->
    tr_expr e [Llir.Return]
```

Les spécificités de ces nouveaux cas sont, pour l'instruction d'affectation l'utilisation d'une fonction auxiliaire traduisant une variable (définie plus bas), et pour l'instruction de retour le fait que l'on néglige l'étiquette `next`, qui n'a plus d'utilité.

Les instructions de contrôle apportent des comportements un peu plus riches, puisqu'il faut des graphes avec des branchements, voire des boucles. Dans le cas d'un branchement conditionnel `if (`c`) { `s_1` } else { `s_2` }`, les deux branches s_1 et s_2 sont traduites avec la même étiquette de suite `next`, qui est le point de jonction. Le bloc évaluant la condition c se conclut par un saut conditionnel, choisissant l'une des deux branches (rappel : `tr_seq` renvoie l'étiquette du bloc de tête de la séquence traduite).

```
  | Imp.If(c, s1, s2) ->
    let then_name = tr_seq s1 next in
    let else_name = tr_seq s2 next in
    tr_expr c [Llir.CJump(then_name, else_name)]
```

Dans le cas d'une boucle, on produit un bloc pour le test de la condition, et un ensemble de blocs pour le corps de la boucle. Pour résoudre la circularité dans le graphe à créer, on génère à la main le nom du bloc réalisant le test, puis on l'enregistre de même manuellement dans la table de hachage (pour ce bloc, on n'utilise donc pas notre fonction auxiliaire `new_block`). Une fois ceci mis en place, on produit un bloc contenant une unique instruction de saut vers le bloc réalisant le test, puisque c'est par là qu'il faudra commencer.

```
  | Imp.While(c, s) ->
    let test_name = new_name() in
    let loop_name = tr_seq s test_name in
    Hashtbl.add code test_name (tr_expr c [Llir.CJump(loop_name, next)]);
    [Llir.Jump test_name]
in
```

La fonction `tr_expr` traduit une expression IMP `e` en une séquence d'instructions LLIR. Cette fonction est récursive : pour traduire une expression `Add(`e_1`, `e_2`)` il faudra traduire les deux sous-expressions e_1 et e_2 et rassembler les deux séquences obtenues. En revanche, on veut éviter d'utiliser l'opérateur `@` de concaténation à cet

endroit (il coûte trop cher). À la place on donne à la fonction `tr_expr` un paramètre supplémentaire : une liste `k` d'instructions LLIR qui doit être placée *après* la séquence obtenue en traduisant `e`. Cela revient à fournir en deuxième paramètre à `tr_expr` la liste avec laquelle le résultat de la traduction de `e` doit être « concaténée » (sans utiliser `@`) [1]. Dans des cas comme celui d'une constante ou d'une variable, où la traduction de `e` est réduite à une unique instruction, il suffit donc de placer cette instruction en tête de `k`.

```
let rec tr_expr e k = match e with
  | Imp.Cst(n) -> Llir.Cst n :: k
  | Imp.Get(x) -> Llir.Get (convert_var x) :: k
```

Dans le cas d'un opérateur binaire la situation est un peu plus riche : nous avons deux appels récursifs à `tr_expr`, et il faut donner à chacun la suite `k` adaptée. Considérons le cas de $\texttt{Add}(e_1, e_2)$. Sa traduction était décrite par l'équation $f(\texttt{Add}(e_1, e_2)) = f(e_1)\texttt{;}\ \texttt{push;}\ f(e_2)\texttt{;}\ \texttt{add}$. Ainsi, la suite à donner à $f(e_2)$ est `add` et la suite à donner à $f(e_1)$ est $\texttt{push;}\ f(e_2)\texttt{;}\ \texttt{add}$ (et dans un cas comme dans l'autre, on ajoute encore la suite globale `k`). On traduit donc d'abord l'expression e_2 en lui donnant la suite `add; k`, ce qui nous donne une liste `k'`, puis on obtient la séquence complète en incluant cette liste dans la suite à donner à la traduction de e_1.

```
  | Imp.Add(e1, e2) ->
    let k' = tr_expr e2 (Llir.Add :: k) in
    tr_expr e1 (LLir.Push :: k')
```

Les autres opérateurs binaires sont traités de même (ici sans prendre la peine de nommer la séquence intermédiaire `k'`).

```
  | Imp.Mul(e1, e2) ->
    tr_expr e1 (Llir.Push :: tr_expr e2 (Llir.Mul :: k))
  | Imp.Lt(e1, e2) ->
    tr_expr e1 (Llir.Push :: tr_expr e2 (Llir.Lt :: k))
```

Pour les appels de fonction on passe par une fonction auxiliaire `tr_args` pour produire une séquence d'instructions évaluant et plaçant sur la pile une liste d'arguments. Comme `tr_expr`, cette fonction prend en deuxième paramètre une séquence de suite.

```
  | Imp.Call(f, args) -> tr_args args (Llir.Call f :: k)

and tr_args args k = match args with
  | [] -> k
  | a::args -> tr_args args (tr_expr a (Llir.Push :: k))
in
```

Notez l'appel récursif à `tr_args` pour traduire la queue de la liste d'arguments, qui prend comme suite le code évaluant le premier argument. Autrement dit, l'argument de tête est évalué et placé sur la pile en dernier : les arguments seront bien empilés dans l'ordre du dernier au premier.

Ne manque que la fonction auxiliaire `convert_var` associant chaque identifiant de variable IMP à une description de variable au format LLIR qui distingue variables globales, paramètres formels et variables locales, et dans les deux derniers cas précise un numéro. La seule chose notable dans cette dernière fonction est l'ordre dans lequel on consulte les tables : en cas de collision de noms, les variables locales masquent les paramètres, qui eux-mêmes masquent les variables globales.

```
(* Fonction auxiliaire : premier indice auquel [x] apparaît dans [l] *)
let get_i_opt x l =
  let rec scan i = function
    | [] -> None
    | y::l -> if x = y then Some i else scan (i+1) l
  in
  scan 0 l
in
```

1. Une autre issue consisterait à utiliser une structure de données autre que la liste, avec concaténation en temps constant.

```
  let convert_var x =
    match get_i_opt x Imp.(fdef.locals) with
      | Some i -> Llir.Local i
      | None -> match get_i_opt x Imp.(fdef.params) with
          | Some i -> Llir.Param i
          | None -> Global x
  in
```

La traduction de IMP vers LLIR est maintenant achevée. Nous allons pouvoir poursuivre dans les sections suivantes en allant depuis LLIR vers un langage assembleur réel.

Bonus : interprétation de LLIR

Avec les éléments en place, on peut écrire un interprète pour les programmes LLIR. Cette étape n'est pas utile pour l'écriture de notre compilateur IMP, mais elle donne une manière de tester le traducteur IMP vers LLIR que nous venons de réaliser. En outre, l'écriture d'un interprète est une manière alternative de décrire la sémantique de notre représentation intermédiaire.

La fonction principale d'interprétation s'applique au programme à interpréter ainsi qu'à une liste d'arguments qui sont les arguments à donner à la fonction `main`. Dans cette fonction principale, on crée une table pour les variables globales (ici, elles ne sont pas initialisées, mais on pourrait aussi choisir de toutes les initialiser à zéro), et on déclenche l'appel initial à la fonction `main`. Cet appel utilise une fonction auxiliaire `exec_call` que l'on va définir à l'intérieur de `exec_prog` et qui sera à nouveau utilisée pour chaque appel de fonction ordinaire.

```
let exec_prog prog args =
  let globals = Hashtbl.create (List.length prog.globals) in

  ...

  let main = List.find (fun fdef -> fdef.name = "main") prog.functions in
  exec_call main args
```

La fonction `exec_call` s'applique à la description d'une fonction LLIR et une liste de valeurs à lui passer en arguments, et interprète l'appel de fonction correspondant. `exec_call` est définie comme une fonction interne à `exec_prog`, et a donc accès notamment à la table des variables globales.

Comme `exec_prog`, cette nouvelle fonction commence par mettre en place des tables pour la gestion des variables associées à l'appel, en l'occurrence avec un tableau pour les paramètres effectifs et un tableau pour les variables locales. On crée également à ce niveau une pile (rappel : on a dit dans la description du langage LLIR que chaque appel de fonction travaillait avec sa propre pile) et deux fonctions auxiliaires pour ajouter ou retirer un élément de la pile. Une fois cette mise en place effectuée, la fonction entame l'interprétation du code de la fonction avec une nouvelle fonction auxiliaire `exec_seq`, en commençant pas le bloc identifié comme bloc d'entrée, et en donnant arbitrairement la valeur initiale `0` au registre.

```
  let rec exec_call f args =
    let params = Array.of_list args in
    let locals = Array.make fdef.nb_locals 0 in
    let stack = Stack.create () in
    let push v = Stack.push v stack in
    let pop () = Stack.pop stack in

    ...

    let fdef = List.find (fun fdef -> fdef.name = f) prog.functions in
    let s = Hashtbl.find fdef.code fdef.start in
    exec_seq s 0
  in
```

La fonction `exec_seq` d'interprétation d'un bloc d'instructions est définie avec une fonction `exec_instr` d'interprétation des instructions élémentaires. Toutes deux sont définies comme des fonctions internes à `exec_call`, et ont donc accès à toutes les structures définies ci-dessus pour les variables locales ou globales. Notez de plus que

toutes les structures introduites pour les variables sont mutables, et pourront donc être modifiées par `exec_instr`.

La fonction `exec_instr` interprète une instruction élémentaire. Elle prend en paramètre supplémentaire la valeur courante du registre, et renvoie la valeur du registre mise à jour après exécution (il ne faut pas oublier de renvoyer cette valeur même lorsqu'elle n'est pas modifiée).

```
    let exec_instr i r = match i with
      | Cst n  -> n
      | Push   -> push r; r
      | Add    -> r + pop()
      | Mul    -> r * pop()
      | Lt     -> if r < pop() then 1 else 0
      | Putchar -> print_char(char_of_int r); r
```

Cette fonction est également susceptible d'accéder aux variables ou de les modifier. On traite un cas de lecture et un cas d'écriture pour chaque sorte de variable, pour tenir compte des différentes structures utilisées.

```
      | Get(Global x) -> Hashtbl.find globals x
      | Get(Param k) -> params.(k)
      | Get(Local k) -> locals.(k)
      | Set(Global x) -> Hashtbl.replace globals x r; r
      | Set(Param k) -> params.(k) <- r; r
      | Set(Local k) -> locals.(k) <- r; r
```

Pour traiter un nouvel appel de fonction, on va chercher la définition de fonction correspondant à l'identifiant `f` donné. Cette définition nous permet de connaître la fonction exécuter, mais aussi le nombre de paramètres à lire sur la pile. Une fonction auxiliaire `pop_args` récupère le bon nombre d'arguments sur la pile. On déclenche ensuite l'appel à proprement parler par un appel (récursif) à la première fonction auxiliaire `exec_call`.

```
      | Call(f) ->
        let fdef = List.find (fun fdef -> fdef.name = f) prog.functions in
        let rec pop_args = function
          | 0 -> []
          | n -> pop() :: pop_args (n-1)
        in
        let args = pop_args fdef.nb_params in
        exec_call fdef args
```

On laisse de côté les cas relatifs aux instructions de saut, qui seront interceptés en amont par la fonction `exec_seq` décrite ci-après.

```
      | _ -> assert false
    in
```

Notez une asymétrie de traitement entre le registre d'une part, et la pile ou les variables d'autre part : la fonction `exec_instr` prend en paramètre la valeur du registre et renvoie la valeur mise à jour, alors qu'elle gère les autres éléments à l'aide de structures mutables. On aurait cependant très bien pu faire du registre une nouvelle donnée mutable, ou prendre en argument et renvoyer les autres structures (qui auraient alors pu être immuables). La version asymétrique présentée ici est une coquetterie produisant un code plus compact.

La fonction auxiliaire `exec_seq` est définie au même niveau que `exec_instr`. Comme elle, `exec_seq` prend en paramètre supplémentaire la valeur courante du registre et renvoie la valeur mise à jour. Cette fonction `exec_seq` se concentre sur la gestion de la séquence et sur les instructions `jump`, `cjump` et `return`. Point commun entre ces trois instructions : on ignore les éventuelles instructions venant après, car l'exécution continue avec un autre bloc, voire dans le cas de `return` revient directement au contexte appelant. Notez que si le programme LLIR est bien formé, cette liste d'instructions suivantes ignorées est vide.

```
    let rec exec_seq seq r = match seq with
      | Jump(lab) :: _ ->
```

```
        let s = Hashtbl.find fdef.code lab in
        exec_seq s r
      | CJump(lab1, lab2) :: _ ->
        let lab = if r <> 0 then lab1 else lab2 in
        let s = Hashtbl.find fdef.code lab in
        exec_seq s r
      | Return :: _ -> r
```

Pour toutes les autres instructions, `exec_seq` fait appel à `exec_instr`. On suppose en outre que la séquence à exécuter n'est jamais vide.

```
      | i::s -> exec_instr i r |> exec_seq s
      | [] -> assert false
    in
```

6.5 Architecture cible

La cible ultime de la compilation est la production d'un programme directement exécutable par un ordinateur physique. La nature d'un tel programme est intimement liée à l'architecture de l'ordinateur cible.

Architecture et boucle d'exécution

Dans ce cours on utilisera une architecture simple mais réelle appelée MIPS, qu'on trouve notamment dans des puces embarquées. Les deux principaux composants de l'architecture sont :

- un *processeur*, comportant un petit nombre de registres qui permettent de stocker quelques données directement accessibles, ainsi que des unités de calcul qui opèrent sur les registres, et
- une grande quantité de *mémoire*, où sont stockées à la fois des données et le programme à exécuter lui-même.

On a deux unités de base pour la mémoire : l'*octet*, une unité universelle désignant un groupe de 8 bits, et le ***mot mémoire***, qui représente l'espace alloué à une donnée « ordinaire », comme un nombre entier. En MIPS, le mot mémoire vaut 4 octets : les données manipulées sont donc généralement représentées par des séquences de 32 bits. En conséquence, un ***entier machine***, c'est-à-dire un nombre entier primitif manipulé par ce processeur est compris entre -2^{31} et $2^{31}-1$ (ou entre 0 et $2^{32}-1$ dans le cas d'entiers « non signés »).

L'architecture MIPS comporte 32 registres, pouvant chacun stocker un mot mémoire. Ces données stockées dans les registres sont directement accessibles à l'unité de calcul.

La mémoire principale a une étendue de 2^{32} octets. Chaque octet de la mémoire est associé à une ***adresse***, qui est un entier entre 0 et $2^{32}-1$, et c'est exclusivement en utilisant ces adresses que l'on accède aux éléments stockés en mémoire. Les données étant organisées sur des mots mémoire de 4 octet, l'architecture impose en outre que les adresses des données ordinaires soient des multiples de 4 (il existe quelques exceptions). L'accès à la mémoire est relativement coûteux : une seule opération de lecture ou d'écriture en mémoire a un coût nettement supérieur à une opération arithmétique travaillant directement sur les registres.

Le programme à exécuter est une séquence d'instructions directement interprétables par le processeur. Chaque instruction est codée sur 4 octets, et l'ensemble est stocké de manière contiguë dans la mémoire.

Un registre spécial `pc` (qui ne fait pas partie des 32) contient l'adresse de l'instruction courante (*program counter*, ou *pointeur de code*). L'exécution d'un programme procède en répétant le cycle suivant :

1. lecture d'un mot mémoire à l'adresse `pc` (*fetch*),
2. interprétation des octets lus comme une instruction (*decode*),
3. exécution de l'instruction reconnue (*execute*),
4. mise à jour de `pc` pour passer à l'instruction suivante (par défaut : incrémenter de 4 octets pour passer au mot mémoire suivant).

Instructions

On distingue trois catégories principales d'instructions :

— des instructions arihtmétiques, appliquant des opérations élémentaires à des valeurs stockées dans des registres,
— des instructions d'accès à la mémoire, pour transférer des valeurs de la mémoire vers les registres ou inversement,
— des instructions de contrôle, pour gérer le pointeur de code `pc`.

Dans les 32 bits utilisés pour coder une instruction, les 6 premiers désignent l'opération à effectuer (***opcode***), et les suivants donnent les paramètres ou des précisions éventuelles sur l'opération. Ci-dessous, quelques exemples avant d'aborder à la section suivante la manière de programmer avec ces instructions.

Par exemple : l'instruction numéro 9 prend en paramètres un registre source r_s, un registre de destination r_t, et un nombre n non signé de 16 bits, et place dans r_t la valeur $r_s + n$. Il suffit de 5 bits pour désigner l'un des 32 registres. Si r_s est le registre numéro 5, r_t le registre numéro 17, et n la valeur 42, alors l'instruction est représentée par un mot mémoire décomposé ainsi :

op	r_s	r_t	n
001000	00101	10001	0000000000101010

Autre exemple : l'instruction numéro 15 prend en paramètres un registre de destination r_t et un nombre n de 16 bits, et place n dans les deux octets supérieurs de r_t. Si r_t est le registre numéro 17 et n la valeur 42, alors l'instruction est représentée par le mot mémoire suivant, où 5 bits ne sont pas utilisés.

op		r_t	n
001111	00000	10001	0000000000101010

Quizz : comment utiliser les instructions précédentes pour charger la valeur `0xeff1cace` *dans le registre numéro 2 ?*

La plupart des opérations arithmétiques binaires utilisent l'*opcode* 0, l'opération précise étant alors précisée dans les 6 derniers bits de l'instruction (cette dernière partie est appelée la *fonction*). Ces instructions prennent invariablement trois paramètres : deux registres r_s et r_t pour les valeurs auxquelles appliquer l'opération, et un registre de destination r_d. Ainsi l'opération d'addition, qui est identifiée par l'*opcode* 0 et la fonction 32. Ci-dessous, opération d'addition avec r_s, r_t et r_d les registres de numéros respectifs 1, 2 et 3. Notez que 5 bits sont inutilisés.

op	r_s	r_t	r_d		f
000000	00001	00010	00011	00000	100000

Certaines opérations ignorent le paramètre r_s et fournissent à la place une donnée constante à l'aide des 5 bits qui étaient restés libres ci-dessus. Ainsi, l'opération de décalage vers la gauche prend trois paramètres : un registre r_t à qui appliquer un décalage, un registre r_d de destination, et une constante k sur 5 bits indiquant de combien de bits vers décaler r_t. C'est opération correspond toujours à l'*opcode* 0, mais cette fois avec la fonction 0. Ainsi, pour décaler de le registre numéro 4 de 5 bits vers la gauche et placer le résultat dans le registre numéro 6, on a :

op		r_t	r_d	k	f
000000	00000	00100	00110	00101	000000

Ce schéma général de découpage vaut encore pour les instructions d'accès à la mémoire. L'instruction de lecture d'un mot mémoire par exemple prend en paramètres deux registres r_s et r_t et un entier signé n sur 16 bits. Le registre r_s est supposé contenir une adresse, et l'entier n est appelé *décalage*. L'instruction de lecture calcule une adresse a en additionnant l'adresse de base r_s et le décalage n, et transfère vers le registre r_t le mot mémoire lu à l'adresse a. Le code de cette opération est 35. Ainsi, pour placer dans le registre numéro 3 la donnée trouvée à l'adresse obtenue en ajoutant 8 octets à l'adresse donnée par le registre numéro 5, on a la séquence :

op	r_s	r_t	n
100011	00101	00011	0000000000001000

Les instructions de contrôle utilisent encore une variante de ce même découpage. On a par exemple une instruction de saut conditionnel, qui prend en paramètres un

registre r_s et un entier n signé sur 16 bits, et qui avance de n instructions dans le code du programme si la valeur de r_s est strictement positive. Le code de cette instruction est 7. Pour reculer de 5 instructions lorsque la valeur du registre 3 est strictement positive, on a donc l'instruction :

op	r_s	r_t	n
000111	00011	00000	1111111111111011

Enfin, l'instruction numéro 2 prend pour unique paramètre un entier sur 26 bits interprété comme l'adresse a d'une instruction i, et va faire se poursuivre l'exécution à partir de cette instruction i. Autrement dit, on écrase la valeur du pointeur `pc` pour la remplacer par a. Pour aller à l'instruction d'adresse `0x00efface` on aura donc l'instruction :

op	a
000010	00111011111111101011001110

Nous verrons à la section suivante qu'en pratique, on ne compose pas directement ces codes représentant chaque instruction. On utilise à la place une forme textuelle appelée le ***langage assembleur***, qui est ensuite automatiquement traduite en la version codée par des mots binaires.

Autres familles d'architectures

De nombreux aspects de l'architecture MIPS que nous venons de décrire sont partagés avec les autres architectures majeures. Pour commencer, la boucle d'exécution *fetch/decode/execute* avec un programme stocké en mémoire est une réalisation directe du modèle de von Neumann, à la base de tous les ordinateurs depuis l'origine de l'informatique.

D'une famille d'architectures à l'autre, on observe des variations par exemple sur la taille d'un mot mémoire, avec notamment les architectures 64 bits où le mot mémoire est de 8 octets. Le nombre de registre peut également varier, ainsi que l'ensemble des instructions disponibles ou les manières d'accéder à la mémoire. Les différences les plus significatives ne sont en revanche pas toujours les plus visibles pour le programmeur.

Dans l'architecture MIPS, les instructions sont codées suivant un schéma uniforme qui limite le nombre d'instructions qui peuvent être proposées. Cette approche est typique des architectures de la famille *RISC* (*Reduced Instruction Set Computer*), qui comprennent notamment les architectures ARM.

À l'inverse, les architectures de la famille *CISC* (*Complex Instruction Set Computer*) proposent un codage variable des instructions, par exemple allant de 1 à 8 octets. Ceci permet entre autres choses de proposer un nombre beaucoup plus importants d'instructions. Cette famille comprend notamment les architectures Intel.

Dans une représentation variable de type *CISC*, on dispose d'instructions riches permettant un code optimisé. En outre, le format variable permet de compresser le code, en donnant une représentation compacte aux instructions les plus fréquemment utilisées. Dans une représentation uniforme de type *RISC*, le décodage des instructions est plus simple, et l'ensemble du processeur est de même plus simple et utilise moins de transistors, ce qui permet une moindre consommation d'énergie. En conséquence les architectures ARM sont omniprésentes dans les *smartphones*, où la gestion de la batterie est critique. Les architectures Intel, réputées plus puissantes, ont à l'inverse longtemps été hégémoniques dans le domaine des ordinateurs grand public, mais cela pourrait basculer. En particulier, le format uniforme des architectures RISC facilite grandement le traitement de plusieurs instructions en parallèle, ce qui est une source très importante d'optimisation des architectures. La montée en puissance de ces optimisations rend maintenant les architectures RISC compétitives avec les architectures CISC en termes de puissance de calcul, et les puces ARM commencent à prendre une place dans le monde des ordinateurs grands publics [2].

2. Ce commentaire est écrit en 2021. Apple est en train de remplacer les processeurs Intel par des processeurs ARM dans l'ensemble de ses ordinateurs, avec succès jusque là (gros gain énergétique sans perte notable de puissance de calcul), et des rumeurs suggèrent que Google préparerait une évolution similaire.

6.6 Langage assembleur MIPS

En pratique, on n'écrit pas directement les suites de bits du langage machine. On utilise à la place un langage d'assemblage, ou assembleur, qui est quasiment isomorphe au langage machine mais permet une écriture plus agréable. En langage assembleur on a en particulier :

- des écritures textuelles pour les instructions,
- la possibilité d'utiliser des étiquettes symboliques plutôt que des adresses explicites,
- une allocation statique des données globales,
- quelques *pseudo*-instructions, qui correspondent à une combinaison simple d'instructions réelles.

Dans cette section, on présente le langage assembleur MIPS.

Pour tester nos programmes MIPS, on utilisera le simulateur MARS. Ce simulateur peut être appelé directement depuis la ligne de commande pour exécuter un programme assembleur donné, mais dispose aussi d'un mode graphique dans lequel on peut suivre pas à pas l'exécution d'un programme et l'évolution des registres et de la mémoire.

Registres

Dans l'assembleur MIPS, les 32 registres sont désignés par leur numéro, de `$0` jusqu'à `$31`. Alternativement à son numéro, chaque registre possède en outre un nom reflétant sa fonction.

n°	*nom*
`$0`	`$zero`
`$1`	`$at`
`$2`	`$v0`
`$3`	`$v1`
`$4`	`$a0`
`$5`	`$a1`
`$6`	`$a2`
`$7`	`$a3`

n°	*nom*
`$8`	`$t0`
`$9`	`$t1`
`$10`	`$t2`
`$11`	`$t3`
`$12`	`$t4`
`$13`	`$t5`
`$14`	`$t6`
`$15`	`$t7`

n°	*nom*
`$16`	`$s0`
`$17`	`$s1`
`$18`	`$s2`
`$19`	`$s3`
`$20`	`$s4`
`$21`	`$s5`
`$22`	`$s6`
`$23`	`$s7`

n°	*nom*
`$24`	`$t8`
`$25`	`$t9`
`$26`	`$k0`
`$27`	`$k1`
`$28`	`$gp`
`$29`	`$sp`
`$30`	`$fp`
`$31`	`$ra`

Les 24 registres `$v*`, `$a*`, `$t*` et `$s*` sont des registres ordinaires, que l'on peut librement utiliser pour des calculs (on verra les spécificités de ces quatre familles plus tard). Les 8 autres registres ont des rôles particuliers : `$gp`, `$sp`, `$fp` et `$ra` contiennent des adresses utiles, `$zero` contient toujours la valeur 0 et ne peut pas être modifié, et les 3 registres `$at` et `$k*` sont réservés respectivement pour l'assembleur et pour le système (il ne faut donc pas les utiliser).

Instructions et pseudo-instructions

Un programme en langage assembleur MIPS prend la forme d'un fichier texte, où chaque ligne contient une instruction ou une annotation. On introduit une liste d'instructions avec l'annotation

```
.text
```

Les instructions ainsi introduites sont placées dans une zone de la mémoire réservée au programme, tout en bas de la mémoire (c'est-à-dire aux adresses les plus basses).

Chaque instruction est construite avec un mot clé appelé ***mnémonique***, désignant l'opération à effectuer, et un certain nombre de paramètres séparés par des virgules. Voici par exemple une instruction qui initialise le registre `$t0` avec la valeur 42

```
li $t0, 42
```

et une instruction qui copie la valeur présente dans le registre `$t0` vers le registre `$t1`.

```
move $t1, $t0
```

Comme le montrent ces exemples, les paramètres donnés aux instructions peuvent être, selon les instructions, des registres désignés par leur nom et/ou des constantes entières écrites de manière traditionnelle.

Une mnémonique décrit l'opération à effectuer, en général à l'aide d'un mot ou d'un acronyme. On a ici `move` pour le « déplacement » d'une valeur ou `li` pour *Load Immediate*, c'est-à-dire pour le chargement d'une valeur constante (on appelle valeur *immédiate* une valeur constante directement fournie dans le code assembleur). Certaines de ces mnémoniques correspondent directement à des instructions machines, et peuvent être associées à un *opcode* et le cas échéant à une fonction. D'autres mnémoniques sont des *pseudo-instructions*, c'est-à-dire des raccourcis pour de courtes séquences d'instructions machine (généralement des séquences d'une ou deux instructions seulement).

En l'occurrence :

— `li` est une pseudo-instruction qui s'adapte à la constante à charger : si cette constante s'exprime sur 16 bits alors elle sera traduite par une unique instruction machine, mais dans le cas contraire on aura deux instructions pour charger indépendamment les deux octets hauts et les deux octets bas. Ainsi, cette pseudo-instruction permet au programmeur de s'affranchir de la limite de 16 bits pour les valeurs immédiates.

— `move` est une pseudo-instruction, traduite par une unique instruction d'addition : plutôt que d'inclure dans les circuits du processeur une instruction spécifique pour une affectation de la forme `$k` ← `$k'`, on réutilise l'instruction d'addition qui est de toute façon présente, en fixant le deuxième opérande à zéro : `$k` ← (`$k'` + `$zero`).

Arithmétique

Les opérations arithmétiques et logiques du langage assembleur MIPS suivent toutes le même format « trois adresses » : après la mnémonique identifiant l'opération viennent dans l'ordre le registre cible (où sera placé le résultat de l'opération) puis les deux opérandes.

```
<mnemo> <dest>, <r1>, <r2>
```

On y trouve des opérations variées, dont la plupart vous seront familières.

— Opérations arithmétiques : `add`, `sub`, `mul`, `div`, `rem` (*remainder* : reste), ...

— Opérations logiques : `and`, `or`, `xor` (*exclusive or*), ...

— Comparaisons : `seq` (*equal* : égalité), `sne` (*not equal* : inégalité) `slt` (*less than* : $<$), `sle` (*less or equal* : $\leqslant$), `slt` (*greater than* : $>$), `sge` (*greater or equal* : $\geqslant$), ...

À ces opérations binaires s'ajoutent quelques opérations unaires classiques également, avec un registre de destination et un seul opérande : `abs` (valeur absolue), `neg` (opposé), **`not`** (négation logique).

Les opérations arithmétiques admettent parfois des variantes pour l'arithmétique non signée, c'est-à-dire pour manipuler des entiers entre 0 et $2^{32} - 1$ plutôt que des entiers entre -2^{31} et $2^{31} - 1$. Ces opérations sont repérables par la présence d'un `u`. Par exemple : `addu`.

Un certain nombre des opérations précédentes admettent également une variante dans laquelle le deuxième opérande est une valeur immédiate plutôt qu'un registre. Ces opérations sont repérables par la présence d'un `i`. Par exemple : `addi`, ou encore `addiu`.

Enfin, les opérations de manipulation des séquences de bits par décalage ou rotation suivent le même format, mais en prenant par défaut une valeur immédiate pour le deuxième opérande (l'amplitude du décalage).

— Décalages et rotations : `sll` (*shift left logical*), `sra` (*shift right arithmetic*), `srl` (*shif right logical*), `rol` (*rotation left*), `ror` (*rotation right*), ...

Des variantes avec une amplitude de décalage variable existent, repérées par la présence d'un `v`. Par exemple : `sllv`. Dans ce cas, le deuxième opérande est un registre.

Voici par exemple une séquence d'instructions pour évaluer la comparaison `3*4 + 5 < 2*9`, ainsi qu'un tableau montrant l'évolution des différents registres à mesure que l'exé-

cution progresse.

	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7	$t8
`li $t0, 3`	3								
`li $t1, 4`	3	4							
`li $t2, 5`	3	4	5						
`li $t3, 2`	3	4	5	2					
`li $t4, 9`	3	4	5	2	9				
`mul $t5, $t0, $t1`	3	4	5	2	9	12			
`add $t6, $t5, $t2`	3	4	5	2	9	12	17		
`mul $t7, $t3, $t4`	3	4	5	2	9	12	17	18	
`slt $t8, $t6, $t7`	3	4	5	2	9	12	17	18	1

L'exemple précédent n'est cependant guère judicieux : il utilise un nouveau registre pour chaque opération, et n'utilise pas toujours les opérations les plus efficaces. Voici une meilleure version pour réaliser le même calcul.

	$t0	$t1
`li $t0, 3`	3	
`li $t1, 4`	3	4
`mul $t0, $t0, $t1`	12	4
`addi $t0, $t0, 5`	17	4
`li $t1, 9`	17	9
`sll $t1, $t1, 1`	17	18
`slt $t0, $t0, $t1`	1	18

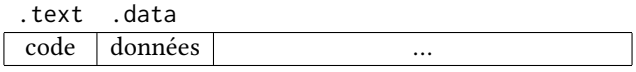
Données statiques

À côté des instructions, un programme assembleur peut également déclarer et initialiser un certain nombre de données, correspondant par exemple à des variables globales du programme.

On introduit une liste de déclarations de telles données avec l'annotation

```
.data
```

En mémoire, les données ainsi déclarées sont placées dans une zone située après la zone réservée aux instructions du programme.



La définition d'une donnée ou d'un groupe de données combine :

- la déclaration d'une étiquette symbolique, c'est-à-dire d'un nom qui pourra être utilisé pour accéder à la donnée,
- la fourniture d'une ou plusieurs valeurs initiales.

Un mot-clé permet de préciser la nature des données fournies : `.word` pour une valeur 32 bits (4 octets), `.byte` pour une valeur d'un unique octet (8 bits), `.asciiz` pour une chaîne de caractères au format ASCII (un caractère par octet, la chaîne étant terminée par l'octet zéro (`0x00`).

On déclare ainsi une donnée désignée par le nom `reponse` et valant 42 :

```
reponse: .word 42
```

Ainsi, un mot mémoire va être réservé dans la zone des données et initialisé avec la valeur 42. L'identifiant `reponse` désigne l'adresse de ce mot mémoire, et peut être utilisée notamment pour récupérer ou modifier la valeur qui y est stockée.

On peut de même introduire une séquence de données.

```
prems: .word 2 3 5 7 11 13 17 19
```

La première donnée est placée à l'adresse correspondant à l'étiquette `prems`, puis les suivantes sont l'une après l'autre, de 4 octets en 4 octets.

L'adresse réservée pour chaque donnée sera calculée au moment de la traduction du programme assembleur en langage machine. À cette occasion, chaque mention d'une étiquette sera remplacée en dur par un accès à l'adresse correspondante.

Accès à la mémoire

Le format standard des adresses mémoire en MIPS contient deux composantes :
— une adresse de base, donnée par la valeur d'un registre `$r`,
— un décalage, donné par une constante `d` (16 bits, signée).
On note `d($r)` une telle adresse, dont la valeur est donc `$r + d`.

Les instructions d'accès à la mémoire permettent alors de transférer une valeur depuis une adresse mémoire vers un registre (lecture, *load*) ou au contraire depuis un registre vers une adresse mémoire (écriture, *store*).

```
lw $t0, 8($a1)
sw $t0, 0($t1)
```

Les deux instructions précédentes enchaînent donc la lecture du mot mémoire à l'adresse obtenue en ajoutant 8 à la valeur du registre `$a1` (la valeur lue étant placée dans le registre `$t0`), puis l'écriture de la valeur du registre `$t0` à l'adresse donnée directement par la valeur du registre `$t1`.

Les mnémoniques `lw` et `sw` signifient respectivement *Load Word* et *Store Word*. Des variantes existent pour lire ou écrire seulement un octet (`lb`, `lbu`, `sb`), ou un demi-mot (`lh`, `lhu`, `sh`), ou encore un double mot (`ld`, qui transfère les 8 octets lus vers 2 registres : celui donné dans l'instruction et le suivant, et `sd` qui fonctionne symétriquement).

L'assembleur permet également quelques notations simplifiées des adresses, qui sont converties vers le format standard au moment de la traduction vers le langage machine. On peut par exemple omettre le décalage lorsqu'il vaut zéro

```
sw $t0, ($t1)
```

ou encore accéder directement à l'adresse donnée par une étiquette, avec un éventuel décalage positif ou négatif.

```
lw $t0, prems
lw $t1, prems+4
lw $t2, prems+12
```

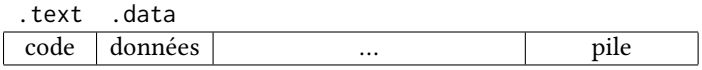
Enfin, une instruction `la` (*Load Address*) permet de récupérer ou calculer une adresse, sans lire son contenu.

```
la $t0, prems
```

Pile

Le mécanisme des données statiques n'est pas suffisant dans un programme ordinaire [3]. On a généralement besoin de pouvoir stocker des données en mémoire, ne serait-ce que temporairement, sans avoir prédit précisément dès l'écriture du programme la quantité d'espace occupée. Ainsi, dans un programme ordinaire la plus grande partie de la mémoire est *dynamique*, c'est-à-dire que l'allocation de l'espace aux différentes données n'y est décidée que *pendant l'exécution* du programme, et qu'une adresse occupée par une donnée à un moment de l'exécution a vocation à être libérée lorsque la donnée n'est plus utile, pour être ensuite réaffectée à une nouvelle donnée.

La partie supérieure de cette grande région de mémoire dynamique est appelée la ***pile***, et fonctionne de manière identique à la structure de données de pile. Il s'agit donc d'une structure linéaire, dont une extrémité est appelée le ***fond*** et l'autre le ***sommet***, et qui est modifiée uniquement du côté de son sommet : on peut ajouter de nouveaux éléments au sommet de la pile, ou retirer les éléments du sommet. Cette organisation implique qu'un élément retiré de la pile est toujours l'élément de la pile le plus récemment ajouté.



Petite particularité notable ici : la pile est placée à l'extrémité de la mémoire, du côté des adresses les plus hautes. Il n'est donc pas possible de l'étendre « vers le haut ».

3. On s'astreint parfois à s'en contenter, pour limiter les possibilités de bugs dans des catégories particulières de programmes, notamment dans des domaines critiques comme l'avionique.

Le fond de cette pile est donc calé sur l'adresse la plus grande de la mémoire, et le sommet à une adresse inférieure : la pile croît ainsi en s'étendant vers les adresses inférieures.

Le sommet de la pile est donc l'adresse mémoire la plus basse utilisée par la pile. Cette adresse est stockée dans le registre `$sp`. On réaliser les opérations usuelles d'une structure de pile de la manière suivante.

— Pour consulter la valeur au sommet de la pile (peek), on lit un mot à l'adresse donnée par `$sp`.

```
lw $t0, 0($sp)
```

Pour aller consulter des éléments plus profonds dans la pile, on ajoute un décalage de 4 octets pour chaque élément à sauter. On consulte donc le quatrième élément avec un décalage de 12 octets.

```
lw $t0, 12($sp)
```

— Pour ajouter un élément au sommet de la pile (push), on décrémente `$sp` de 4 octets, puis on écrit la valeur souhaitée à la nouvelle adresse `$sp`.

```
addi $sp, $sp, -4
sw  $t0, 0($sp)
```

— Pour retirer l'élément au sommet de la pile, on incrémente `$sp` de 4 octets. Notez qu'il n'est pas besoin « d'effacer » la valeur présente à l'ancienne adresse `$sp` : le simple fait que cette adresse soit maintenant inférieure à `$sp` la désigne comme insignifiante.

```
addi $sp, $sp, 4
```

L'opération usuelle pop, consistant à récupérer l'élément au sommet tout en le retirant de la pile, combine cet incrément avec une opération peek.

```
lw $t0, 0($sp)
addi $sp, $sp, 4
```

Le code assembleur obtenu est d'une certaine manière symétrique à celui de push.

Sauts et branchements

Rappelons que les instructions d'un programme MIPS sont stockées en mémoire. Chaque instruction a donc une adresse. De même que les adresses des données, les adresses de certaines instructions peuvent être désignées par des étiquettes : une étiquette insérée dans une séquence d'instructions MIPS désigne l'adresse de l'instruction qui la suit immédiatement, et pourra être utilisée comme cible d'une instruction de saut.

MIPS propose une variété d'instructions (ou pseudo-instructions) de branchement conditionnel. En voici un exemple :

```
beq $t0, $t1, lab
```

Cette instruction compare les valeurs des registres `$t0` et `$t1`. Si ces deux valeurs sont égales, alors l'exécution se poursuivra avec l'instruction d'étiquette `lab` (techniquement, l'instruction de branchement commande une modification du registre spécial pc). Sinon, l'exécution se poursuivra naturellement avec l'instruction suivante.

Des variantes existent pour d'autres modalités de comparaison

$=$	$\neq$	$<$	$\leqslant$	$>$	$\geqslant$
beq	bne	blt	ble	bgt	bge

et chacune admet à nouveau une variante pour comparer une unique valeur à zéro : beqz, bnez, bltz, ... Certaines admettent également pour deuxième paramètre une valeur immédiate plutôt qu'un registre. Enfin, une pseudo-instruction `b lab` provoque un branchement inconditionnel vers l'instruction d'étiquette `lab`.

Voici un exemple de fragment de code calculant la factorielle du nombre stocké dans le registre `$a0`, et plaçant le résultat dans le registre `$v0`.

```
1        move $t0, $a0
2        li $t1, 1
3        b test
     loop:
4        mul $t1, $t1, $t0
5        addi $t0, $t0, -1
     test:
6        bgtz $t0, loop
7        move $v0, $t1
```

Les deux premières instructions initialisent les deux registres `$t0` et `$t1` qui seront utilisés pour le calcul. Les instructions 4 à 6 forment une boucle, et la dernière instruction transfère le résultat dans `$v0` comme demandé. Le corps de la boucle est constitué des instructions 4 et 5, la première étant associée à l'étiquette `loop`. Le test de la boucle est à l'instruction 6, associée à l'étiquette `test`. Lorsque ce test est positif, on déclenche un nouveau tour de boucle à l'aide d'un branchement vers `loop`. À l'inverse, lorsque le test est négatif on poursuit avec l'instruction suivante, c'est-à-dire l'instruction 7. Enfin, l'instruction 3 démarre la boucle en branchant vers le test. *Notez que l'on aurait pu se passer de* `$t0`*, de* `$t1` *et des instructions 1 et 7 en travaillant exclusivement avec* `$a0` *et* `$v0`*.*

Les instructions de branchement sont utilisées pour des déplacements « locaux » dans la séquence d'instructions, c'est-à-dire en restant dans la même unité de code (par exemple : la même fonction). Les instructions machine correspondantes font avancer ou reculer le pointeur de code d'un certain nombre d'instructions, ne dépassant pas 2^{15}.

À l'inverse, les instructions de sauts définissent la prochain instruction de manière absolue, en fournissant directement son adresse. Cela peut se faire à l'aide d'une étiquette, et dans ce cas la différence avec une instruction de branchement n'est pas flagrante,

```
j label
```

ou en fournissant directement une adresse calculée, stockée dans un registre.

```
jr $t0
```

Les instructions de saut servent notamment dans le cadre d'un saut non local, comme un appel de fonction. Pour cette situation, on dispose également de deux instructions qui, avant de sauter, sauvegardent une adresse de retour dans le registre `$ra`, qui permettra plus tard de revenir à l'exécution de la séquence en cours.

```
jal label
jalr $t0
```

On reviendra sur ce mécanisme d'appel de fonction à la section suivante.

Appels système

Pour certaines fonctionnalités dépendant du système d'exploitation, un code assembleur doit faire appel aux bibliothèques système. Dans ce cours, on exécutera notre code MIPS dans un simulateur, qui prendra en charge ces différents services avec une instruction dédiée. On décrit donc ici des fonctionnalités du simulateur, qui miment des services dépendant normalement du système d'exploitation.

On déclenche un appel système à l'aide de l'instruction `syscall`, après avoir placé dans le registre `$v0` un code désignant le service demandé. Voici une petite sélection :

service	code	arg.	rés.
affichage	1 (entier) 4 (chaîne) 11 (ascii)	`$a0`	
lecture	5 (entier)		`$v0`
arrêt	10		
extension mémoire	9	`$a0`	`$v0`

On trouve encore des services pour la manipulation de fichiers, et d'autres variantes d'affichage ou de lecture.

Hello, world

Le programme le plus célèbre de la terre, en MIPS.

```
        .text
main:   li $v0, 4
        la $a0, hw
        syscall
        li $v0, 10
        syscall

        .data
hw:     .asciiz "hello world\n"
```

Notes : `4` est le code de l'appel système d'affichage d'une chaîne de caractères, `10` celui de l'arrêt. L'annotation `.asciiz` permet de placer une chaîne de caractères dans les données statiques, dont l'adresse est ici associée à l'étiquette `hw`.

6.7 Fonctions et conventions d'appel

Un point clé du développement de programmes réalistes est la possibilité de définir et d'appeler des fonctions. Du point de vue de la compilation, cela demande un peu de technologie supplémentaire. Fixons d'abord le vocabulaire, en observant l'***appel de fonction*** `f(6)` dans le code suivant.

```
1   void g() {
2     int y;
3     y = f(6);
4     print(y);
5   }

6   int f(int x) {
7     return x*7;
8   }
```

La valeur `6` est le ***paramètre effectif*** (on dit aussi l'*argument*) de l'appel de fonction `f(6)`. Le bloc de code allant des lignes 2 à 4 est le ***contexte appelant***, et la fonction `f` définie aux lignes 6 à 8 est la ***fonction appelée***. Du côté de la fonction appelée, la variable `x` est le ***paramètre formel*** de la fonction, et l'expression `x*7` calcule le résultat ***renvoyé*** par la fonction.

Appel de fonction : mécanisme de base

Un appel de fonction implique des transerts de différents natures entre le contexte appelant et la fonction appelée. On a d'une part un transfert de données, avec

- un ou plusieurs paramètres effectifs transmis par le contexte appelant à la fonction appelée,
- un résultat, renvoyé au contexte appelant par la fonction appelée.

En MIPS, on convient usuellement que les paramètres effectifs sont transmis par les registres `$a0` à `$a3` et que le résultat renvoyé est transmis par le registre `$v0`. *Notez que c'est cohérent avec les appels système vus ci-dessus.* Dans le cas où il y a plus de quatre paramètre effectifs, on utilise les registres `$a*` pour les quatre premiers paramètres, puis la pile pour les suivants.

Outre ce transfert de données, on a un transert *temporaire* de contrôle.

- Lors de l'appel, l'exécution *saute* au code de la fonction appelée.
- À la fin de l'appel, l'exécution *revient* au contexte appelant, en reprenant « juste après » le point où on a réalisé l'appel.

Ce transfert temporaire demande, au moment de l'appel, de mémoriser l'adresse de l'instruction à laquelle il faudra revenir après l'appel. En MIPS, on utilise pour cela le registre `$ra` (*Return Address*). Pour sauter au code de la fonction tout en mémorisant dans `$ra` l'adresse de l'instruction suivante du contexte appelant, on utilise

```
jal f
```

À la fin de l'appel de fonction, on rend alors la main à l'appelant avec au saut à l'adresse donnée par `$ra`.

```
jr $ra
```

Voici une traduction possible du fragment de code précédent, où les nombres à gauche donnent l'adresse de chaque instruction.

```
12  g:  li $a0, 6
16      jal f
20      move $a0, $v0
24      li $v0, 1
28      syscall

36  f:  li $v0, 7
40      mul $v0, $a0, $v0
44      jr $ra
```

Note : pour obtenir des adresse réalistes il faut ajouter à chacune la constante `0x400000`. Voici une trace d'exécution de ce code assembleur.

instruction	$a0	$v0	$ra	pc	
				12	
`li $a0, 6`	6			16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42

Observons maintenant ce qui se passe si on ajoute un code principal réalisant un appel à notre fonction `g`. On complète le code comme suit.

```
00  main: jal g
04        li $v0, 10
08        syscall
12  g:    li $a0, 6
16        jal f
20        move $a0, $v0
24        li $v0, 1
28        syscall
32        jr $ra
36  f:    li $v0, 7
40        mul $v0, $a0, $v0
44        jr $ra
```

La trace d'exécution est maintenant la suivante.

instruction	$a0	$v0	$ra	pc	
				0	
`jal g`			4	12	
`li $a0, 6`	6		4	16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42
`jr $ra`	42	1	20	20	
`move $a0, $v0`	1	1	20	24	
`li $v0, 1`	1	1	20	28	
`syscall`	1	1	20	32	affiche 1
`jr $ra`	1	1	20	20	
`move $a0, $v0`	1	1	20	24	
…	…	…	…	…	

Nous voilà coincés dans une boucle ! Problème : lorsque l'instruction `jal f` d'appel à `f` stocke son adresse de retour dans `$ra`, elle écrase la valeur qui était déjà présente

dans `$ra`, à savoir l'adresse de retour de l'appel à g. Ainsi, lorsque l'on arrive à l'instruction `jr $ra` qui achève le code de g à l'adresse 32, le registre `$ra` ne contient plus la valeur qui avait été sauvegardée au moment de l'appel à g, et notre saut se fait vers la mauvaise cible.

Pile d'appels

Lorsque des appels de fonction sont emboîtés, c'est-à-dire lorsque le code d'une fonction appelée réalise lui-même un appel de fonction et ainsi de suite, on se retrouve avec plusieurs appels actifs à un même moment. Plus précisément, l'appel le plus récent est en cours d'exécution et un certain nombre d'autres appels plus anciens sont en suspens.

Chaque appel de fonction actif possède son propre contexte, contenant notamment les arguments qui lui ont été passés, les valeurs de ses variables locales, ou encore l'adresse de retour stockée dans `$ra`. Toutes ces informations doivent être stockées de manière à survivre à des appels de fonction internes. La structure utilisée pour stocker les informations d'un appel donné est appelée ***tableau d'activation*** (en anglais *call frame*). Un tableau d'activation a la forme générale suivante :

variables locales	registres sauvegardés	arguments

où la partie « registres sauvegardés » contient en particulier la valeur sauvegardée du registre `$ra`.

Ces tableaux sont stockés dans la mémoire. On utilise un registre dédié `$fp` (*Frame Pointer*) pour localiser le tableau actif, c'est-à-dire le tableau d'activation de l'appel actuellement en cours d'exécution. On peut ensuite, à l'aide de `$fp`, accéder aux différents éléments stockés dans le tableau. Le pointeur de base `$fp` désigne l'adresse du dernier mot dédié aux registres sauvegardés.

		`$fp`
variables locales	registres sauvegardés	arguments

On accède donc aux arguments à partir de `$fp` avec un décalage positif, et aux variables locales avec un décalage négatif.

La vie des tableaux d'activation se déroule ainsi :

— création d'un tableau d'activation au début d'un appel,

— destruction du tableau à la fin de l'appel.

En outre, tout appel doit se terminer avant de rendre la main à son contexte appelant. Autrement dit, c'est toujours l'appel le plus récent qui terminera en premier, et toujours le tableau d'activation le plus récent qui sera détruit en premier. L'ensemble des tableaux d'activation a donc une structure de pile (*Last In First Out*), nommée ***pile d'appels*** (*call stack*). La pile d'appels est réalisée dans la pile MIPS elle-même, et en forme l'ossature principale.

Ainsi le premier mot du tableau d'activation correspond au sommet de la pile, désigné par le pointeur `$sp`.

`$sp`		`$fp`
variables locales	registres sauvegardés	arguments

En outre, pour faciliter l'identification du tableau d'activation précédent (qui est celui du contexte appelant), tout tableau d'activation a dans ses registres sauvegardés la valeur précédente du registre de base `$fp`, c'est-à-dire l'adresse qui permet d'accéder à ce tableau précédent.

Convention d'appel

Pour assurer les bons transferts d'information et de contrôle, et la bonne gestion de la pile d'appels, le contexte appelant et la fonction appelée doivent respecter un protocole commun appelé ***convention d'appel***. Ce protocole se découpe en plusieurs étapes, à la charge de l'un ou l'autre des participants :

1. L'appelant, avant l'appel : prend en charge la passation des arguments via des registres et/ou la pile, puis déclenche l'appel avec `jal` pour stocker l'adresse de retour dans `$ra`.
2. L'appelé, au début de l'appel : réserve de l'espace sur la pile pour le tableau d'activation, y sauvegarde les registres qui doivent l'être, et donne à `$fp` sa nouvelle valeur.

3. Exécution du corps de la fonction appelée.
4. L'appelé, à la fin de l'appel : place le résultat dans un registre dédié, restaure les registres qui avaient été sauvegardés, désalloue le tableau d'activation, et redonne la main à l'appelant avec `jr $ra`.
5. L'appelant, après l'appel : retire les arguments placés sur la pile.

Le tableau d'activation est donc créé à l'étape 2 et détruit à l'étape 4 : il existe uniquement pendant l'exécution du corps de la fonction appelée.

Dans la convention standard de MIPS, les quatre premiers arguments sont passés par les registres `$a0` à `$a3` et les suivants éventuels par la pile, et le résultat est renvoyé par le registre `$v0`. Il est possible d'utiliser une convention différente, à condition de rester cohérent, et que tout appel respecte bien une même convention. *On utilisera effectivement une convention simplifiée pour compiler le langage LLIR, notamment en passant l'intégralité des arguments pas la pile (rappel : ce langage intermédiaire utilise uniquement une pile et un accumulateur).*

Dans la convention MIPS usuelle, on a également une subtilité au niveau de la sauvegarde des registres. Les registres sont découpés en deux familles :

— la sauvegarde des registres `$s*` est à la charge de l'appelé (*callee-saved*), de même que ce que nous avons déjà vu pour `$ra` et `$fp`,

— la sauvegarde des registres `$a*` et `$t*` est à la charge de l'appelant (*caller-saved*).

Autrement dit, dans l'étape 1 du protocole d'appel, l'appelant doit également sauvegarder les valeurs des registres `$a*` ou `$t*` dont il aurait encore besoin après l'appel (et il peut ensuite les restaurer à l'étape 5). Inversement, l'appelé n'a besoin à l'étape 2 de sauvegarder que les valeurs des registres `$ra`, `$fp` et `$s*`, et même plus précisément que les valeurs des registres qu'il modifiera. En conséquence, on utilise les registres `$t*` en priorité pour des valeurs intermédiaires à la durée de vie courte (pour ne pas avoir besoin de les sauvegarder avant de réaliser un appel). À l'inverse, les éléments que l'on souhaite préserver sur une durée plus longue seront avantageusement stockés dans les registres `$s*` : ils n'auront alors à être sauvegardés que lorsqu'une fonction appelée aura effectivement besoin d'utiliser ces mêmes registres. *À nouveau, cette subtilité sera invisible dans notre compilateur du langage LLIR, puisque ce dernier n'utilise presque pas de registres.*

Code MIPS pour une fonction récursive

Imaginons une fonction factorielle, définie par un code de la forme suivante.

```
int fact(int n) {
  if (n > 0) {
    return fact(n-1) * n;
  } else {
    return 1;
  }
}
```

Après traduction en assembleur MIPS suivant les conventions précédentes, on obtient une fonction attendant un paramètre passé par le registre `$a0` et renvoyant un résultat via le registre `$v0`. Voici donc un fragment de code MIPS réalisant l'appel `fact(10)` et affichant le résultat.

```
       # Appel
00     li     $a0, 10
04     jal    fact
       # Affichage
08     move   $a0, $v0
12     li     $v0, 1
16     syscall
       # Fin du programme
20     li     $v0, 10
24     syscall
```

Le code de la fonction elle-même est construit comme suit. D'abord, on réserve de l'espace sur la pile pour le tableau d'activation, dans lequel on sauvegarde les registres `$ra` et `$fp`. Notez que l'on a prévu la place dans le tableau d'activation pour un troisième élément, qui servira plus tard.

```
   fact:
       # Tableau d'activation
28     sw     $fp, -4($sp)
32     sw     $ra, -8($sp)
36     addi   $fp, $sp, -4
40     addi   $sp, $sp, -12
```

Ensuite vient le corps de la fonction à proprement parler, commençant par un test. Si le test est positif, on saute au fragment de code correspondant à la branche « then ». Sinon on poursuit avec l'instruction suivante, correspondant à la branche « else ».

```
       # Test
44     bgtz   $a0, fact_rec
48     li     $v0, 1
52     b      fact_end
  fact_rec:
```

Dans la branche « then » il faut réaliser un appel récursif à `fact`, avec un argument décrémenté de 1. Or la valeur de l'argument servira encore après l'appel : il faudra la multiplier avec le résultat de l'appel. On sauvegarde donc cette valeur avant de préparer l'appel récursif.

```
       # Appel recursif
56     sw     $a0, -8($fp)
60     addi   $a0, $a0, -1
64     jal    fact
```

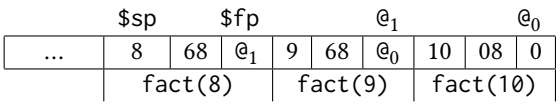
Après l'appel, on récupère donc la valeur sauvegardée pour réaliser la multiplication.

```
68     lw     $t0, -8($fp)
72     mul    $v0, $v0, $t0
```

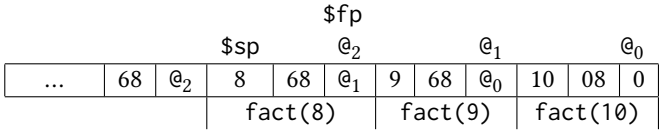
Enfin, on ajoute une section finale s'occupant de conclure l'appel de fonction. Notez que l'on arrive à cette section soit après la branche « then », soit après la branche « else », mais que dans un cas comme dans l'autre le résultat a déjà été placé dans `$v0`. Il ne reste donc ici qu'à restaurer les registres `$fp` et `$ra` et détruire le tableau d'activation.

```
  fact_end:
       # Nettoyage
76     addi   $sp, $fp, 4
80     lw     $fp, -4($sp)
84     lw     $ra, -8($sp)
88     jr     $ra
```

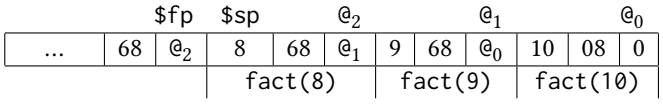
En abordant la ligne 28 au début de l'appel récursif `fact(7)`, la pile d'appels a la forme suivante :



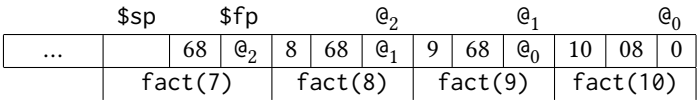
Les lignes 28 et 32 écrivent les valeurs courantes de `$fp` et `$ra` juste au-delà du sommet de la pile.



La ligne 36 déplace le pointeur `$fp` juste au-delà du sommet de la pile.



Enfin, la ligne 40 complète la création du tableau d'activation en déplaçant le sommet de pile. La case destinée à recevoir la valeur de `$a0` est présente, mais n'est pas encore initialisée.



Les lignes 76 à 84 effectuent l'opération inverse et nous ramènent à la configuration initiale.

Appels terminaux

Regardons le cas où une fonction termine par un appel à une autre fonction (on parle d'un ***appel terminal***).

```
int f() {
  return g(3);
}
```

Cette fonction `f` n'a plus rien à faire après l'appel à `g`, à part nettoyer son propre tableau d'activation. Le tableau d'activation de l'appel à `f` n'est donc plus utile au moment de l'appel à `g`, et on pourrait imaginer le détruire *avant* l'appel. En outre, la valeur de `$ra` sauvegardée dans le tableau d'activation de l'appel à `f`, qui est restaurée lors de la destruction du tableau d'activation, est alors précisément l'adresse à laquelle il faudra retourner après l'appel à `g`. On peut donc réaliser l'appel avec un simple saut, c'est-à-dire une instruction `j` plutôt que `jal`. Et pour limiter encore les manipulations on peut, au lieu de détruire le tableau d'activation de l'appel à `f`, le réutiliser pour l'appel à `g` (il faut affiner les détails si les deux fonctions utilisent des tableaux d'activation de tailles différentes).

Cette optimisation des appels terminaux a un effet particulièrement spectaculaire lorsqu'on l'applique à une fonction récursive.

```
int fact(int n) {
  return fact_aux(n, 1);
}

int fact_aux(int n, int acc) {
  if (n <= 0) {
    return acc;
  } else {
    return fact_aux(n-1, n*acc);
  }
}
```

On conserve les lignes 00 à 24 à l'identique pour appeler `fact` et afficher le résultat. Le reste va être simplifié drastiquement.

La fonction `fact` réalise un unique appel terminal à la fonction `fact_aux`. Il n'y a pas de tableau de tableau d'activation à créer, le premier argument est déjà dans `$a0`, il suffit de placer le deuxième argument dans `$a1` puis sauter à la fonction auxiliaire.

```
   fact:
28'    li    $a1, 1
32'    j     fact_aux
```

Notez que l'on utilise l'instruction de saut `j` et pas l'instruction d'appel `jal`, puisque `$ra` a déjà la bonne valeur.

Vient ensuite la fonction auxiliaire `fact_aux`. Celle-ci n'a pas non plus besoin de tableau d'activation, puisqu'elle n'a pas de variables locales et n'opère que des appels terminaux à elle-même. On réalise donc directement le test. S'il est positif on saute à la branche « then », qui sera le cas d'arrêt.

```
   fact_aux:
36'    blez  $a0, fact_aux_end
```

Sinon on poursuit avec l'appel récursif terminal, pour lequel il suffit de placer les bonnes valeurs dans `$a0` et `$a1`.

```
40'    mul   $a1, $a1, $a0
44'    addi  $a0, $a0, -1
48'    j     fact_aux
```

Enfin la fonction termine en plaçant le résultat dans `$v0` et en revenant, enfin, à l'adresse de retour enregistrée lors de l'appel d'origine à `fact`, ligne 4.

```
   fact_aux_end:
52'    move  $v0, $a1
56'    jr    $ra
```

Le code obtenu est beaucoup plus simple, et utilise exclusivement les registres. À noter : il est même identique à ce qu'on aurait pu obtenir en compilant un code impératif basé sur une boucle comme la suivante.

```
int res = 1;
while (n > 0) {
  res = n*res;
  n--;
}
```

6.8 Traduction LLIR vers MIPS

Reprenons maintenant notre langage LLIR, pour le traduire en assembleur MIPS. La combinaison de cette traduction avec la précédente fournira donc un compilateur de IMP vers MIPS.

On suit la stratégie générale suivante :

— L'unique registre de LLIR est matérialisé par le registre `$t0` en MIPS. On appelle ce registre l'accumulateur.

— La pile LLIR est réalisée avec la pile MIPS et le pointeur `$sp`.

On utilisera très peu d'autres registres MIPS : un registre auxiliaire `$t1` pour les opérations arithmétiques et certaines opérations mémoire, plus les registres spéciaux `$sp`, `$fp` et `$ra` (le registre `pc` est toujours utilisé également, mais on ne le manipule pas explicitement).

LLIR étant déjà bas niveau, la plupart des instructions LLIR se traduisent en seulement une ou deux instructions MIPS. Le plus gros travail concerne la réalisation des fonctions et des appels. Pour cela, on va se laisser guider par les conventions vues pour MIPS, avec de petites simplifications reflétant l'usage qui était fait dans LLIR de la pile et de l'accumulateur.

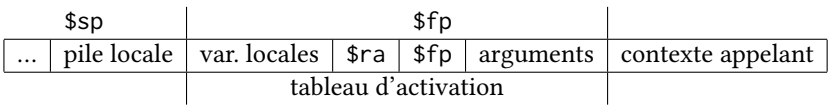
— Tous les arguments sont placés sur la pile, avec le premier argument au sommet (repris de LLIR).

— Le résultat d'un appel est placé dans l'accumulateur `$t0` (repris de LLIR).

— À charge pour l'appelé de sauvegarder et restaurer `$fp` et `$ra` (MIPS standard).

— Les autres registres n'étant pas utilisés, on n'a jamais besoin de les sauvegarder ou de les restaurer au moment des appels.

Lors d'un appel de fonction la pile aura donc la forme suivante.



On utilise toujours le registre `$fp` pour l'adresse de référence du tableau d'activation de l'appel en cours. Au-dessus de cette adresse de référence on trouve l'intégralité des paramètres effectifs (et pas juste les paramètres à partir du cinquième). À l'adresse de référence et à l'adresse immédiatement inférieure on stocke l'ancienne valeur du pointeur `$fp` et l'adresse de retour `$ra`. Sous ces deux mots, le tableau d'activation est complété par une zone dédiée aux variables locales. En dessous du tableau d'activation, on se sert de la pile MIPS et de son pointeur `$sp` pour réaliser la pile locale de l'appel de fonction LLIR.

Module auxiliaire MIPS

Pour représenter le code MIPS, on se donne un type minimal permettant de représenter efficacement des concaténations de chaînes de caractères.

```
type asm =
  | Nop
  | S of string
  | C of asm * asm

let (@@) x y = C (x, y)

type program = { text: asm; data: asm; }
```

Ainsi, pour « concaténer » deux fragments a_1 et a_2 de code assembleur, il suffit de former la structure `C(`a_1`,` a_2`)`, ce que l'on peut abréger avec la notation a_1 `@@` a_2. C'est plus efficace qu'une vraie concaténation, qui aurait un coût proportionnel à la longueur des chaînes manipulées.

Pour faciliter la production de code MIPS sous cette représentation, on définit en plus de cela quelques constantes pour les registres utilisés

```
let t0 = "$t0"
let t1 = "$t1"
let ra = "$ra"
let sp = "$sp"
let fp = "$fp"
```

ainsi qu'une série de fonctions auxiliaires produisant des instructions MIPS. L'objectif est que le code caml produisant une représentation de code assembleur MIPS soit aussi proche que possible du code MIPS représenté. On pourra ainsi écrire en caml dans du code caml l'expression `add t0 t0 t1` pour inclure la chaîne de caractères `"  add $t0, $t0, $t1\n"` dans un élément de type `asm`.

Voici les premières de ces fonctions.

```
open Printf
let li   r1 i     = S(sprintf "  li   %s, %i"  r  i)
let move r1 r2    = S(sprintf "  move %s, %s"  r1 r2)

let add  r1 r2 r3 = S(sprintf "  add  %s, %s, %s" r1 r2 r3)
let addi r1 r2 i  = S(sprintf "  addi %s, %s, %d" r1 r2 i)
let mul  r1 r2 r3 = S(sprintf "  mul  %s, %s, %s" r1 r2 r3)
let slt  r1 r2 r3 = S(sprintf "  slt  %s, %s, %s" r1 r2 r3)

let j    l        = S(sprintf "  j    %s"      l)
let jal  l        = S(sprintf "  jal  %s"      l)
let jr   r1       = S(sprintf "  jr   %s"      r1)
let bnez r1 l     = S(sprintf "  bnez %s, %s"  r1 l)
let bltz r1 l     = S(sprintf "  bltz %s, %s"  r1 l)
let bge  r1 r2 l  = S(sprintf "  bge  %s, %s, %s" r1 r2 l)

let syscall       = S("  syscall")
let nop           = Nop
let label l       = S(sprintf "%s:" l)
let comment s     = S(sprintf "  # %s" s)
```

Pour refléter la possibilité des instructions MIPS de la famille de `lw` et `sw` de décoder des adresses sous différents formats, on introduit également une distinction entre deux formats utiles : une adresse donnée directement par une étiquette, et une adresse calculée par un décalage à partir de l'adresse donnée par un registre.

```
type address =
  | L of string
  | A of int * string

let (++) i r = A(i, r)

let address = function
  | L l    -> l
  | A(i, r) -> sprintf "%i(%s)" i r

let lw  r1 a    = S(sprintf "  lw   %s, %s" r1 (address a))
```

```
let sw  r1 a    = S(sprintf "  sw   %s, %s" r1 (address a))
let lbu r1 a    = S(sprintf "  lbu  %s, %s" r1 (address a))
```

Pour traiter les déclarations de données statiques, on introduit encore une les fonctions auxiliaires suivantes.

```
let ilist = function
  | [] -> ""
  | [i] -> sprintf "%i" i
  | i :: l -> sprintf "%i, %s" i (ilist l)
let dword l = S(sprintf "  .word %s" (ilist l))
let asciiz s = S(sprintf "  .asciiz %s" s)
```

Cette batterie de fonctions auxiliaires donne une manière fluide de générer en caml une représentation de code MIPS. Par exemple, le code MIPS ci-dessous à gauche peut être défini par l'expression caml à droite.

```
  # built-in atoi
atoi:
  li    $v0, 0
  li    $t1, 10
atoi_loop:
  lbu   $t0, 0($a0)
  beqz  $t0, atoi_end
  addi  $t0, $t0, -48
  bltz  $t0, atoi_error
  bge   $t0, $t1 atoi_error
  mul   $v0, $v0, $t1
  add   $v0, $v0, $t0
  addi  $a0, $a0, 1
  j     atoi_loop
atoi_error:
  li    $v0, 10
  syscall
atoi_end:
  jr    $ra
```

```
let built_ins =
  comment "built-in atoi"
  @@ label "atoi"
  @@ li    v0 0
  @@ li    t1 10
  @@ label "atoi_loop"
  @@ lbu   t0 (0++a0)
  @@ beqz  t0 "atoi_end"
  @@ addi  t0 t0 (-48)
  @@ bltz  t0 "atoi_error"
  @@ bge   t0 t1 "atoi_error"
  @@ mul   v0 v0 t1
  @@ add   v0 v0 t0
  @@ addi  a0 a0 1
  @@ j     "atoi_loop"
  @@ label "atoi_error"
  @@ li    v0 10
  @@ syscall
  @@ label "atoi_end"
  @@ jr    ra
```

Pour finir, on ajoute deux fonctions push et pop produisant des séquences de code MIPS facilitant la manipulation de la pile.

— push *r* génère du code MIPS qui empile le contenu du registre *r*, et
— pop *r* génère du code MIPS qui retire l'élément au sommet de la pile et le place dans le registre *r*.

Ces deux éléments n'existent pas directement dans les instructions ou pseudo-instructions MIPS. Via l'utilisation de notre module MIPS on pourra les voir comme des *pseudo*-pseudo-instructions.

```
let push r =
  addi sp sp (-4) @@ sw r (0++sp)
let pop r =
  lw r (0++sp) @@ addi sp sp 4
```

Cette représentation caml des programmes MIPS étant fixée, une simple fonction permet ensuite d'écrire l'ensemble d'un programme MIPS vers une destination cible, par exemple un fichier.

```
let rec print_asm fmt a =
  match a with
  | Nop -> ()
  | S s -> fprintf fmt "%s\n" s
  | C (a1, a2) ->
     let () = print_asm fmt a1 in
     print_asm fmt a2

let print_program fmt p =
  fprintf fmt ".text\n";
```

```
  print_asm fmt p.text;
  fprintf fmt ".data\n";
  print_asm fmt p.data
```

Traduction de LLIR vers MIPS

La fonction de traduction principale `tr_prog` s'applique à un programme LLIR `prog`, et renvoie un programme MIPS. La fonction de traduction des instructions `tr_instr` est définie comme fonction interne (elle aura besoin d'accéder à la liste `prog.functions`).

```
open Llir
open Mips

let tr_prog prog =
  let tr_instr = function
    | Cst n -> li t0 n
```

Rappel : LLIR place les valeurs calculées dans un accumulateur (le registre `$t0`), et les résultats intermédiaires sur la pile. Pour tout ce qui concerne l'arithmétique on utilise donc principalement ce registre et les deux opérations auxiliaires `push` et `pop`. Juste avant de réaliser une opération binaire il faut récupérer la valeur du premier opérande, qui a été stockée sur la pile une fois calculée. On utilise pour cela l'opération `pop`, et un deuxième registre de travail `$t1`.

```
    | Push -> push t0
    | Add  -> pop t1 @@ add t0 t1 t0
    | Mul  -> pop t1 @@ mul t0 t1 t0
    | Lt   -> pop t1 @@ slt t0 t1 t0
```

Un accès à une variable en lecture (`Get`) ou en écriture (`Set`) est directement traduit par une instruction `lw` ou `sw`. Il faut cependant calculer la localisation de cette variable, qui varie en fonction de sa nature :

- les variables globales sont allouées dans le segment des données statiques, avec une étiquette à leur nom,
- les variables locales sont stockées dans le tableau d'activation, avec un décalage négatif par rapport à l'adresse de base `$fp`,
- les paramètres d'un appel de fonction sont stockés dans le tableau d'activation, avec un décalage positif par rapport à l'adresse de base `$fp`.

Une fonction auxiliaire `loc` produit ces localisations

```
let loc = function
  | Global x -> (L x)
  | Param i -> (4*(i+1))++fp
  | Local i -> (-4*(i+2))++fp
```

et peut être directement utilisée dans les cas correspondants de la fonction `tr_instr`.

```
    | Get(v) -> lw t0 (loc v)
    | Set(v) -> sw t0 (loc v)
```

Les sauts LLIR sont traduits par des sauts MIPS. Pour représenter un branchement conditionnel à deux branches tel que je `CJump` de LLIR, on utilise en MIPS un saut conditionnel vers la première branche, suivi d'un saut vers la deuxième branche (exécuté si le premier branchement n'a pas été sélectionné).

```
    | Jump lab -> j lab
    | CJump(lab1, lab2) -> bnez t0 lab1 @@ j lab2
```

L'instruction d'affichage `putchar` est traduite par un appel système. Pour respecter les règles de `syscall` il faut donc transférer la valeur à afficher de l'accumulateur `$t0` vers le registre `$a0`.

```
    | Putchar -> move a0 t0 @@ li v0 11 @@ syscall
```

Dans la traduction d'un appel de fonction `Call`, il faut inclure les étapes du protocole d'appel qui sont à la charge de l'appelant. En l'occurrence il n'y a plus rien à faire

avant l'appel (on suppose que les arguments ont déjà été placés au sommet de la pile), mais il faut encore se charger de nettoyer la pile après l'appel. Pour cela on consulte la définition de la fonction appelée pour connaître le nombre d'arguments à retirer de la pile.

```
    | Call f ->
      let fdef = List.find (fun fdef -> fdef.name = f) prog.functions in
      let n = fdef.nb_params in
      jal f
      @@ addi sp sp (4*n)
```

L'instruction `return` appartient au domaine de l'appelé. C'est à ce niveau que l'on prend en charge la partie du protocole d'appel devant être réalisée par l'appelé, à la fin de l'appel. Notez qu'à ce stade le résultat est déjà dans l'accumulateur `$t0`, il ne reste donc qu'à désallouer le tableau d'activation et à restaurer les registres `$ra` et `$fp`.

```
    | Return ->
      addi sp fp (-4)
      @@ pop ra
      @@ pop fp
      @@ jr ra
  in
```

Pour traduire une séquence d'instructions, il suffit d'itérer la fonction précédente.

```
  let rec tr_seq s =
    List.fold_right (fun i asm -> tr_instr i @@ asm) s nop
  in
```

La traduction d'une fonction LLIR contient deux aspects :

— il faut traduire l'ensemble des blocs composant le code de cette fonction, et
— il faut ajouter un code d'introduction qui réalise la partie du protocole d'appel qui est à la charge de l'appelé, au début de l'appel, pour ensuite passer la main à la première « vraie » instruction de la fonction.

Du côté de la convention d'appel, on a besoin de sauvegarder `$fp` et `$ra`, définir la nouvelle valeur de `$fp` et allouer le tableau d'activation. Cette partie est précédée de l'étiquette de la fonction, qui sera utilisée comme cible de saut lors d'un appel, et suivie du saut vers la première instruction du corps de la fonction.

```
  let tr_fdef fdef =
    label fdef.name
    @@ push fp
    @@ push ra
    @@ addi fp sp 4
    @@ addi sp sp (-4*fdef.nb_locals)
    @@ j fdef.start
```

On peut ensuite placer dans un ordre arbitraire l'ensemble des blocs composant la fonction, chacun précédé de son étiquette.

```
    @@ Hashtbl.fold
      (fun name seq code -> code @@ label name @@ tr_seq seq)
      fdef.code nop
  in
```

On obtient le texte complet du programme en combinant le code de chacune des fonctions, y compris la fonction prédéfinie `atoi` vue ci-dessus, avec un bloc de code principal qui :

— regarde si le paramètre `arg` de la fonction `main` a reçu une valeur sur la ligne de commande, et dans ce cas le traduit en un entier à l'aide de notre fonction MIPS auxiliaire `atoi`,
— appelle la fonction `main`,
— termine l'exécution du programme une fois l'appel à `main` achevé.

```
  let text =
      beqz a0 "init_end"
```

```
    @@ lw  a0 0 a1
    @@ jal "atoi"
    @@ sw  v0 (0++sp)
    @@ addi sp sp (-4)
    @@ label "init_end"
    @@ jal "main"
    @@ li  v0 10
    @@ syscall

    @@ List.fold_right
        (fun fdef asm -> tr_fdef fdef @@ asm)
        prog.functions
        nop

    @@ built_ins
  in
```

Ne reste plus qu'à y ajouter les déclarations des variables globales, allouées dans les données statiques et initialisées avec la valeur `0`.

```
  let data =
    List.fold_right
      (fun x asm -> labal x @@ dword [0] @@ asm)
      prog.globals
      nop
  in

  { text; data }
```

6.9 Tableaux

Jusqu'ici, nous n'avons compilé que de petits langages de programmation, dans lesquels les données manipulées étaient limitées à des nombres entiers. L'objectif de la fin de ce chapitre est d'y ajouter des structures de données plus riches.

Les données, selon leur nature, peuvent être représentées de différentes manières en mémoire :

- un nombre entier ou un booléen est représenté par un unique mot mémoire (par exemple de 4 octets),
- une donnée plus complexe comme un tableau, un n-uplet, une structure, un objet ou une fermeture sera en revanche représenté par plusieurs mots, formant généralement un *bloc*, c'est-à-dire une suite de mots consécutifs en mémoire,
- une donnée plus riche encore comme une liste chaînée, un arbre ou une table de hachage pourra même être représentée par plusieurs blocs, non consécutifs en mémoire.

Tableau alloué statiquement

Considérons le cas d'un tableau global, dont la taille est explicitement donnée par le texte du programme source. Dans un langage comme C on pourrait par exemple introduire ainsi une variable globale `tab` représentant un tableau de cinq entiers.

```
int tab[5];
void main(int x) { ... }
```

Comme les autres variables globales vues jusqu'ici, ce tableau peut être placé en mémoire dans les données statiques. En revanche, au lieu de lui allouer un unique mot mémoire, on en donne cinq consécutifs.

```
      .data
tab:  .word 0 0 0 0 0
```

Le premier de ces mots est à l'adresse donnée par l'étiquette `tab`, le suivant à l'adresse `tab+4`, et le dernier à l'adresse `tab+16`. Notez que l'on ne peut faire une telle déclaration que si l'on connaît explicitement la taille du tableau.

L'accès à une case d'un tel tableau se fait donc en partant de l'étiquette `tab`, qui donne une adresse de base, et en ajoutant un décalage correspondant à l'indice de la case cherchée (le décalage étant un multiple de 4 octets).

Considérons par exemple le tableau introduit par

```
      .data
tab:  .word 1 1 2 5 14 42
```

On accède au troisième élément à l'aide d'un décalage de 8 par rapport à l'adresse de base donnée par `tab` :

```
      la   $t0, tab
      lw   $t1, 8($t0)
```

Dans cette première version l'indice est connu statiquement : on peut calculer le décalage en amont, et l'écrire explicitement dans le code. Si en revanche l'indice auquel chercher est donné par un registre, il va falloir ajouter un peu de code MIPS pour calculer à l'exécution le bon décalage. Pour accéder à une case dont l'indice est donné par le registre `$a0`, il faut donc multiplier la valeur de `$a0` par 4 pour obtenir le bon décalage (on le fait efficacement par un décalage de 2 bits vers la gauche, avec l'instruction `sll`), puis ajouter ce décalage à l'adresse de base.

```
      la   $t0, tab
      sll  $a0, 2
      add  $t0, $t0, $a0
      lw   $t1, 0($t0)
```

Lors de l'instruction `lw` on n'indique donc plus de décalage par rapport à l'adresse donnée par `$t0`, puisque que celui-ci contient déjà précisément l'adresse voulue. *Note : le décalage donné dans l'instruction* `lw` *ne peut être qu'une constante entière explicite, on ne peut pas y mettre* `$a0`.

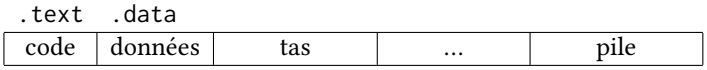
Tableaux alloués dynamiquement

Imaginons cette fois le cas d'un tableau créé dynamiquement, avec une taille qui n'est pas connue explicitement à l'avance.

```
int f(int n) {
  int tab[n];
  ...
}
```

C'est beaucoup plus souple pour le programmeur, mais demande une nouvelle stratégie pour le compilateur. On peut imaginer que ce tableau soit représenté par n mots consécutifs en mémoire, rangés sur la pile avec les autres variables locales. C'est effectivement ce qui se passe en C, par défaut. Attention cependant : comme tout ce qui se trouve sur la pile, ce tableau a une durée de vie limitée et sera détruit à la fin de l'appel.

Nous allons nous concentrer ici sur une troisième stratégie, permettant d'allouer dynamiquement des structures de données *persistantes*, qui vont survivre à l'appel de fonction qui les a créées. On utilise pour cela une nouvelle région de la mémoire : le *tas*, qui se trouve entre les données statiques et la pile.



La plus petite adresse du tas est immédiatement au-dessus de la zone des données statiques. Un pointeur `brk` appelé *memory break* identifie la fin du tas, ou plus précisément la première adresse au-delà du tas. Ainsi le dernier mot du tas est à l'adresse `brk-4`. L'espace situé entre le tas et la pile, c'est-à-dire entre les pointeurs `brk` (inclus) et `$sp` (exclu), est vide. Lorsque l'on a besoin de plus de place sur le tas, on incrémente le pointeur `brk`, pour absorber une partie de l'espace libre entre le tas et la pile.

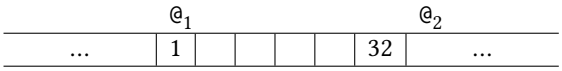
Le pointeur `brk` est géré par le système d'exploitation, aussi son déplacement est fait par un appel système `sbrk`, qui a le code 9 dans notre simulateur MIPS. Cet appel système prend comme argument, via `$a0`, le nombre d'octets dont `brk` doit être incrémenté. Il renvoie comme résultat, via `$v0`, l'ancienne valeur de `brk`, c'est-à-dire la première adresse de la zone qui vient d'être ajoutée au tas.

Voici un exemple d'utilisation de ce mécanisme, dans lequel on alloue sur le tas de l'espace pour un tableau de 6 cases (24 octets), avant d'écrire `1` dans la première

case et `32` dans la sixième.

instruction	`$a0`	`$v0`	`$t0`	`brk`
				$0x10040000$
`li $a0, 24`	24			$0x10040000$
`li $v0, 9`	24	9		$0x10040000$
`syscall`	24	$0x10040000$		$0x10040018$
`li $t0, 1`	24	$0x10040000$	1	$0x10040018$
`sw $t0, 0($v0)`	24	$0x10040000$	1	$0x10040018$
`li $t0, 32`	24	$0x10040000$	32	$0x10040018$
`li $t0, 20($v0)`	24	$0x10040000$	32	$0x10040018$

La forme du tas associée à cet exemple est la suivante, où l'adresse $@_1$ est la position d'origine de `brk` (0x10040000) et $@_2$ est la nouvelle position après appel à `sbrk` (0x10040018).



Extension de LLIR

Pour l'instant, ni notre langage source IMP ni notre représentation intermédiaire LLIR ne contiennent les ingrédients nécessaires pour la manipulation des tableaux. On peut cependant étendre légèrement ces deux langages et les schémas de compilation associés pour obtenir une version améliorée de IMP et de son compilateur.

On ajoute au langage LLIR trois instructions, pour la lecture et l'écriture en mémoire, ainsi que pour l'extension du tas.

— Lecture en mémoire : l'instruction

```
read
```

lit la valeur à l'adresse mémoire donnée par le registre, et place la valeur lue dans le registre. On peut la traduire en MIPS par l'unique instruction

```
lw   $t0, 0($t0)
```

Rappel : dans notre traduction de LLIR vers MIPS, l'unique registre LLIR est réalisé par le registre MIPS `$t0`.

— Écriture en mémoire : l'instruction

```
write
```

écrit la valeur du registre à l'adresse mémoire prélevée au sommet de la pile. On la traduit en MIPS par une séquence retirant l'élément au sommet de la pile puis écrivant la valeur du registre `$t0` à l'adresse ainsi obtenue.

```
lw   $t1, 0($sp)
addi $sp, $sp, 4
sw   $t0, 0($t1)
```

Rappel : dans notre traduction de LLIR vers MIPS, on utilise le registre `$t1` pour travailler sur une valeur extraite de la pile.

— Extension du tas : l'instruction

```
sbrk
```

étend le tas d'un nombre d'octets donné par le registre. Le résultat de l'appel `sbrk`, qui est la première adresse de la zone ajoutée au tas, est placé dans le registre. On traduit cette instruction en MIPS par une séquence effectuant un appel système `sbrk` puis déplaçant son résultat du registre `$v0` où le résultat est placé vers le registre `$t0` où le résultat est attendu.

```
move $a0, $t0
li   $v0, 9
syscall
move $t0, $v0
```

Extension de IMP avec des pointeurs : PIMP

On étend la syntaxe abstraite de IMP avec des constructions correspondant directement aux trois nouvelles instructions LLIR : deux nouvelles formes d'expressions pour la lecture en mémoire et l'extension du tas, et une nouvelle forme d'instruction pour l'écriture en mémoire.

```
type expr =
  ...
  | Read of expr
  | Sbrk of expr

type instr =
  ...
  | Write of expr * expr
```

La traduction de ces nouveaux éléments en LLIR est similaire à ce que l'on a déjà vu pour la traduction des opérations arithmétiques ou de l'instruction `putchar`.

- Une expression `Read` e est traduite en LLIR par :
  - le code évaluant e, qui place la valeur obtenue dans le registre, suivi de
  - l'instruction `read`.
- Une expression `Sbrk` e est traduite de la même façon, avec l'instruction `sbrk`.
- Une instruction `Write(`e_1`,` e_2`)` est traduite en LLIR par :
  - le code évaluant e_1, puis
  - l'instruction `push`, suivie par
  - le code évaluant e_2, et enfin
  - l'instruction `write`.

On étend enfin la syntaxe concrète du langage IMP et les analyseurs associés pour permettre l'utilisation en pratique de ces nouveaux éléments. On introduit les nouvelles formes concrètes suivantes

- `sbrk(1024)` pour l'extension du tas d'un certain nombre d'octets,
- `*p` pour l'accès à l'adresse donnée par un pointeur,
- `t[k]` pour l'accès à une case d'un tableau.

L'accès à une adresse peut être utilisé aussi bien en lecture qu'en écriture. L'analyse syntaxique des expressions va donc reconnaître la forme

```
*p
```

et la traduire sous une forme

```
Read p
```

et l'analyse syntaxique des instructions va reconnaître la nouvelle forme

```
*p = e;
```

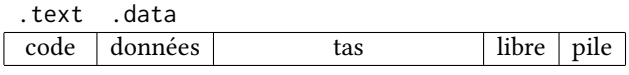
et la traduire sous une forme

```
Write(p, e)
```

La forme concrète `t[k]` n'est en revanche qu'un sucre syntaxique pour `*(t+4k)`. Ainsi une expression de la forme `t[k]` est traduite en la forme `Read(Add(t, Mul(k, Cst 4)))` et une instruction de la forme `t[k] = e` est traduite en la forme `Write(Add(t, Mul(k, Cst 4)), e)`.

6.10 Gestion dynamique de la mémoire

Nous avons évoqué avec les tableaux l'utilisation d'une nouvelle région de la mémoire, appelée le *tas*, que nous allons maintenant détailler.



Le tas est une région de la mémoire qui est adaptée au stockage des données persistantes. Cette région est gérée par un programme dédié qui permet en particulier de :

- demander de la mémoire pour stocker de nouvelles données,
- libérer la mémoire qui avait été affectée à des données qui ne sont plus utiles, pour qu'elle puisse être à nouveau utilisée pour d'autres données.

Ces deux opérations sont qualifiées de *dynamiques*, car elles ont lieu *pendant l'exécution* du programme principal.

Selon les cas, le programme dédié à la gestion de la mémoire peut prendre différentes formes.

— En C, on utilise la bibliothèque `malloc`, et notamment les fonctions `malloc` et `free` pour allouer ou libérer manuellement de la mémoire.
— En caml, java, python, un gestionnaire automatique appelé *GC* (*garbage collector*, ou *glaneur de cellules*) est intégré à l'environnement d'exécution. Il tourne en parallèle du programme principal et identifie puis libère les parties du tas qui ne sont plus utiles.

Nous allons détailler ici le fonctionnement des opérations manuelles `malloc` et `free`, qui peuvent ensuite servir de base à la réalisation d'un gestionnaire automatique.

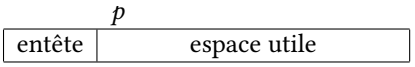
On veut donc deux opérations `malloc` et `free` répondant aux spécifications suivantes.

— Un appel `malloc(`n`)`, avec n un entier positif, trouve et réserve au moins n octets consécutifs dans le tas, et renvoie un pointeur vers le premier octet réservé. Cette fonction ne peut réserver qu'une zone de mémoire qui était libre jusque là. Si la fonction ne trouve pas de bloc mémoire libre assez grand elle peut soit étendre le tas (en absorbant une partie de l'espace libre situé entre le tas et la pile) soit échouer.
— Un appel `free(`p`)` libère le bloc de mémoire à l'adresse donnée par le pointeur p. Un tel appel suppose que le pointeur p cible effectivement un bloc précédemment alloué par `malloc` (sinon, l'effet est indéterminé).

Nous allons d'abord détailler ce qu'est un *bloc mémoire*, puis comment réaliser `malloc` et `free`.

Blocs mémoire

Les blocs sont l'unité de gestion de la mémoire dynamique. Un bloc est une zone de mémoire contiguë, qui peut être libre ou utilisée. Chaque bloc est constitué d'un entête et d'un espace utile.



L'espace utile est la partie effectivement utilisée pour stocker des données, tandis que l'entête contient des informations additionnelles utilisées exclusivement par le gestionnaire de mémoire. On accède au contenu d'un bloc à l'aide d'un pointeur p, qui donne l'adresse du premier octet utile. On a couramment des contraintes sur l'adresse désignée par le pointeur p, par exemple qu'il s'agisse d'un multiple de 8 (on le supposera effectivement dans la suite).

En supposant que l'entête soit représenté par un mot mémoire, et chaque donnée stockée dans l'espace utile soit également représentée par un mot mémoire, on peut donc accèder à l'entête avec `p[-1]` et au mot d'indice i avec `p[`i`]`.

L'entête d'un bloc doit contenir les informations utiles au gestionnaire de mémoire. Les deux principales sont :

— le statut *libre* ou *réservé* du bloc, correspondant à un bit d'information valant `1` pour un bloc réservé et `0` pour un bloc libre,
— la *taille* totale du bloc (espace utile + entête), exprimée par un nombre entier d'octets.

Si l'on suppose que la taille d'un bloc est un multiple de 8, cette dernière est représentée par un nombre dont les trois derniers bits ne donnent pas d'information (pour un multiple de 8 ces trois bits valent nécessairement `0`). On peut alors enregistrer à la fois la taille t et le statut s dans un unique mot mémoire e, en plaçant le bit de statut à la place du dernier bit de taille. On peut réaliser ceci à l'aide d'une opération de « ou » bit à bit : $e = t$ `|` s.

t	s	e
32	libre	$0b100000$
24	réservé	$0b011001$

On peut à l'inverse récupérer le statut ou la taille à partir d'un mot d'entête e à l'aide d'opérations de masquage :

— le statut est donné par le dernier bit : $s = e$ `& 0x1`,
— la taille est obtenue en masquant les trois derniers bits : $t = e$ `& ~0x7`.

Initialisation

On considère à partir de maintenant que l'ensemble du tas est formé de blocs consécutifs en mémoire, sans espace libre entre les blocs (éventuellement cela peut nécessiter de donner un peu plus d'espace utile que demandé à certains blocs, pour bien toujours tomber sur des multiples de 8 pour les adresses).

Pour initialiser un tel tas, on demande au système une zone de mémoire à l'aide d'un appel `sbrk`. Pour marquer les extrémités de cette zone, on laisse une valeur spéciale (en l'occurrence zéro) dans le premier mot et le dernier mot. Tout le reste forme un grand bloc.

		tas		
•	bloc unique		•	`brk`
0	e	espace utile	0	

Si on a alloué 1024 octets pour l'ensemble du tas, une fois sacrifiés les premier et dernier mots il reste 1016 octets pour le bloc : 4 octets d'entête et 1012 octets utiles. Ce bloc est vide, son bit de statut vaut donc 0. Ainsi l'entête e vaut 1016. Notez également que le pointeur p vers l'espace utile de cet unique bloc est un multiple de 8, puisque cette adresse se situe deux mots après le début du tas (lui-même à l'adresse `0x1004000` en MIPS).

Lorsque le tas est étendu par un nouvel appel à `sbrk`, on déplace le zéro final pour préserver la forme générale : une séquence de blocs consécutifs, donc les extrémités sont marquées par deux mots nuls.

				tas						
•	bloc 1		bloc 2		bloc 3		bloc 4		•	`brk`
0	e_1	d_1	e_2	d_2	e_3	d_3	e_4	d_4	0	

Fonction `malloc`

Considérons un appel `malloc(`n`)`. La fonction `malloc` doit rechercher dans le tas un bloc libre suffisamment grand. On peut imaginer une première version qui parcourt séquentiellement tous les blocs du tas jusqu'à en trouver un adapté. Ce parcourt nécessite deux opérations principales.

— Tester un bloc désigné par un pointeur p. Pour cela, il faut regarder l'entête (p`[-1]`), la décomposer en un bit de statut s et une taille t, et vérifier d'une part le statut libre (`!`s), et d'autre part que la taille est bien suffisante pour accueillir n mots dans l'espace utile (n `<=` t `-` 4).
— Passer au bloc suivant. En ajoutant au pointeur p désignant un bloc la taille t de ce bloc, on obtient l'adresse du bloc suivant : on a un chaînage implicite des blocs.

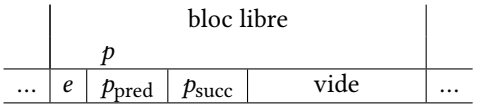
Ces opérations de base étant fixées, on peut varier les stratégies. Par exemple :

— commencer par le premier bloc et s'arrêter au premier bloc convenable trouvé (*first fit*),
— commencer au bloc touché le plus récemment et s'arrêter au premier bloc convenable trouvé (*next fit*), ou
— chercher le plus petit des blocs convenables (*best fit*).

Enfin, si on a parcouru l'ensemble des blocs sans en trouver de convenable on peut effectuer un nouvel appel à `sbrk` pour étendre le tas, du moins tant qu'il reste de l'espace disponible. On réserve alors un nouveau bloc taillé dans cette extension.

Cette première version de la recherche parcourt séquentiellement tous les blocs, libres ou réservés, jusqu'à avoir choisi le bloc à réserver. On pourrait faire nettement plus efficace en ne considérant que les blocs libres. Cela suppose en revanche d'avoir un chaînage entre les blocs libres, c'est-à-dire d'être capable, étant donné un bloc libre, de déterminer efficacement l'adresse d'un bloc libre « suivant ». Une astuce pour cela consiste à remarquer que tout bloc libre est… libre ! Autrement dit, l'espace utile d'un bloc libre n'est pas utilisé par le programme client, et l'allocateur peut l'utiliser pour stocker des méta-données supplémentaires. En l'occurrence, on prend les deux premiers champs de l'espace utile d'un bloc libre pour stocker les adresses

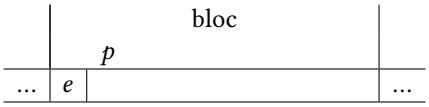
p_{pred} et p_{succ} du bloc libre précédent et du bloc libre suivant.



À noter : pour bien avoir la place de stocker ces deux pointeurs, il faut que la taille de chaque bloc soit d'au moins 12 octets, et même 16 octets en tenant compte de la contrainte « multiple de 8 ».

On peut donc coder, à l'aide des ces zones temporairement inutilisées de la mémoire, une liste doublement chaînée des blocs libres (*free list*), qui sera aisément mise à jour à chaque changement de statut d'un bloc. Il faut simplement mémoriser quelque part, dans une variable globale, l'adresse d'un premier bloc libre. On améliore encore cette technique en n'utilisant pas une seule mais plusieurs listes doublement chaînées de blocs libres, en distinguant plusieurs catégories de tailles (*segregated free lists*).

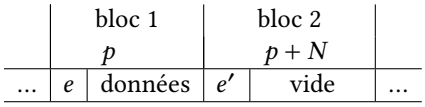
Une fois qu'on a trouvé un bloc convenable, désigné par un pointeur p vers son premier octet utile, on peut le réserver.



Il y a encore deux possibilités :

— on peut réserver intégralement le bloc, en modifiant simplement son bit de statut ($e = e$ `|` `0x1`),

— si le bloc est trop grand, on peut aussi le découper en deux blocs, un qui sera réservé et l'autre laissé libre.



Dans ce schéma, on a réservé un premier bloc de taille N, pour une certaine valeur $N \geq n + 4$ multiple de 8, et laissé un deuxième bloc libre de taille $t - N$.

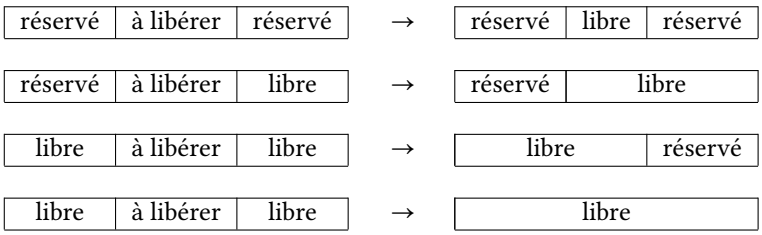
Le choix de découper ou non est à la discrétion de l'allocateur mémoire, et n'est pas si simple a priori : découper permet d'éviter de sous-employer un bloc, mais peut aussi impliquer à terme une fragmentation de la mémoire en de multiples petits blocs difficilement réutilisables. De même, le choix de la valeur N appartient à la stratégie de l'allocateur.

Fonction `free`

Il y a une manière extrêmement simple de libérer un bloc désigné par un pointeur p : aller modifier dans son entête le bit de statut, pour le repasser à 0 ($e = e$ `&` `~0x1`). Dans le cas d'une stratégie *free list* ou *segregated free lists* il faut ajouter à cela l'intégration du bloc libéré dans la liste de blocs libres. On peut le faire par exemple juste avant le premier bloc, en suivant un algorithme standard d'insertion dans une liste doublement chaînée.

Le défaut de cette manière simple est l'apparition progressive de *fragmentation* de la mémoire : si on a découpé un grand bloc pour éviter de perdre de la place (ce qui peut être une bonne chose), on aura après libération plusieurs petits blocs. Ces petits blocs ne pourront être utilisés que pour de petites allocations, et il sera à terme de plus en plus difficile de trouver un bloc de grand taille pour une une allocation d'un grand nombre d'octets.

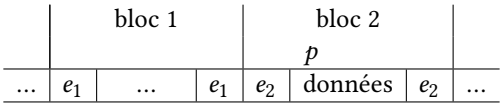
On peut améliorer la situation en fusionnant un groupe de blocs libres adjacents, c'est-à-dire consécutifs en mémoire. Cette opération de fusion peut même être intégrée à l'opération `free` elle-même : si on trouve un bloc libre juste avant ou juste après le bloc libéré par `free`, on les fusionne. Avec les différents statuts « libre » ou « réservé » des deux blocs entourants le bloc libéré par `free`, on a les quatre situations possibles suivantes.



À noter : si ces fusions sont faites systématiquement, on a la garantie de ne jamais avoir deux blocs libres consécutifs.

Pour réaliser ces fusions il faut pouvoir consulter les méta-données, c'est-à-dire les entêtes, des blocs situés juste avant et juste après le bloc libéré.

- Partant du pointeur p du bloc à libérer, on accède facilement au bloc suivant : il suffit d'ajouter la taille du bloc à libérer (et l'entête est le mot précédent l'adresse obtenue).
- En l'état, on n'a en revanche pas de manière directe de calculer l'adresse du bloc précédent. On peut régler ce problème en enrichissant encore un peu la structure de nos blocs, en répétant l'entête d'un bloc à la fin de celui-ci (*boundary tags*).



On accède alors aux méta-données du bloc précédent en regardant deux mots avant l'adresse p.

La technique des *boundary tags* permet de meilleures fusions. Elle a néanmoins un coût : il faut sacrifier un bloc supplémentaire pour cette deuxième copie de l'entête. On peut cependant ajouter une nouvelle optimisation : ne répéter l'entête que dans le cas d'un bloc libre, car c'est seulement dans cette situation que l'on a besoin de la taille, et ajouter dans l'entête de chaque bloc un bit donnant le statut du bloc précédent.

Bilan : un bon allocateur de mémoire est un programme assez subtil, qui repose sur des structures de données qui sont cachées dans les interstices du tas. Il y a beaucoup de choix de stratégie à faire, et une abondance d'études empiriques sur l'efficacité des différents stratégies. Dans Linux, la bibliothèque `malloc.c` *contient plus de 5000 lignes de code.*

6.11 Structures de données

Les langages de programmation permettent couramment la définition de structures de données composées de plusieurs champs, chaque champ étant identifié et accessible par un nom. On peut citer par exemple :

- les structures en C

```
struct point {
  int x;
  int y;
};
```

- les enregistrements en caml

```
type point = {
  x: int;
  y: int;
}
```

- les objets en java

```
class Point {
  public final int x;
  public final int y;
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, `point` désigne un nouveau type de données, `x` et `y` désignent des noms de champs de ce type de données, et les occurrences de `int` désignent le type du contenu associé aux champs `x` et `y`.

Manipulation de structures

On peut représenter une structure en mémoire par un bloc de plusieurs mots mémoire consécutifs, où les valeurs des champs sont stockées l'une après l'autre.

x	y	z	b
1	2	42	true

Dans un type de structure donné, on place systématiquement les champs dans le même ordre. Ainsi, chaque nom de champ peut être associé à une position, et le compilateur traduit chaque accès à un champ identifié par son nom en un accès à la position correspondante dans le bloc mémoire.

$$\begin{aligned} x &\mapsto 0 \\ y &\mapsto 1 \\ z &\mapsto 2 \\ b &\mapsto 3 \end{aligned}$$

On crée une telle structure en allouant l'espace disponible pour le bloc mémoire complet, et en initialisant éventuellement les champs (le détail dépendant du langage). En particulier :

— En caml, on note

```
let p = { x=1; y=2; } in ...
```

la création d'une nouvelle structure `point` dont les champs `x` et `y` sont initialisés respectivement avec 1 et 2. Le bloc est créé dans le tas, ce qui implique une allocation de type `malloc`. La valeur de la variable `p` de type `point` est ensuite concrètement un pointeur vers le bloc alloué dans le tas. On accède au champ `x` de `p` avec la notation `p.x`.

— En java, la situation est similaire à celle de caml : on crée une structure (un objet) avec

```
Point p = new Point(1, 2);
```

Cette opération alloue un bloc sur le tas, et définit `p` comme un pointeur vers ce bloc. On accède au champ `x` de `p` de même avec `p.x`.

— En C, on alloue l'espace pour une structure sur le tas à l'aide de `malloc`.

```
point *p = malloc(sizeof(point));
```

L'adresse du bloc explicitement alloué par `malloc` est ensuite explicitement stockée dans une variable `p` de type `point*`, c'est-à-dire un pointeur vers une structure de type `point`. On accède au champ `x` de la structure pointée par `p` avec la notation `p->x`. On peut donc initialiser ensuite la structure avec les instructions supplémentaires

```
p->x = 1;
p->y = 2;
```

À noter, C admet également la notation utilisée en caml mais avec une signification légèrement différente : `point p = { x=1; y=2; };` place la structure sur la pile plutôt que sur le tas, et la variable `p` désigne directement cette structure et non un pointeur. Et dans ce cas, on accède au champ `x` avec la même notation `p.x`.

Un langage avec structures de données

On se propose d'étendre le langage Mini-C du DM avec des structures de données définies par l'utilisateur. On introduit pour cela les nouveaux éléments de syntaxe concrète suivants :

— Une nouvelle déclaration

```
struct s { t1 x1; ...; tN xN; }
```

pour la définition du type `s` d'une structure dont les champs ont pour noms les `xi` et pour types les `ti`.

— Un nouveau type
  `s*`
  désignant un pointeur vers une structure de nom `s`.
— Deux nouvelles formes d'expressions
  `p->x`
  pour l'accès au champ `x` de la structure pointée par `p`, et
  `sizeof(s)`
  pour le calcul du nombre d'octets à allouer pour une structure de type `s`.
— Une nouvelle instruction
  `p->x = e`
  pour l'affectation d'une nouvelle valeur au champ `x` de la structure pointée par `p`.

La définition d'un programme ne se limite plus à une liste de variables globales et une liste de fonctions : on y ajoute une liste de types de structures.

On peut ensuite compiler ce langage Mini-C, vers le langage PIMP (l'extension de IMP avec des pointeurs décrite à la section 6.9). Ces deux langages sont très proches : ils possèdent la même base impérative, et diffèrent seulement par leur manière d'accéder aux données du tas :

— en Mini-C, on définit des structures de données allouées dans le tas, et on accède à leurs champs via les noms avec `e->x`,
— en PIMP, on manipule directement des pointeurs et des calculs d'adresses.

Pour compiler un accès `e->x` de Mini-C vers PIMP, il faut donc :

1. identifier le type `s` de la structure pointée par `e`,
2. consulter la définition de `s` pour obtenir la position du champ `x` dans la structure,
3. en déduire l'adresse de la donnée à laquelle on veut accéder.

On a ici un phénomène que nous n'avions pas observé dans les traductions de PIMP vers LLIR ou LLIR vers MIPS : on a besoin des types pour traduire un programme Mini-C vers PIMP ! Ici, les types ne servent donc pas seulement à vérifier la cohérence des programmes.

On peut donc suivre l'organisation suivante pour un compilateur de Mini-C :

1. L'analyse syntaxique produit un AST Mini-C dans lequel on trouve, en plus du code, la liste des définitions de types de structures et les types déclarés pour chaque variable ou fonction.
2. Le vérificateur de type prend en entrée un AST Mini-C. Son résultat n'est pas un booléen (bien typé/mal typé), mais un nouvel AST « Mini-C typé ». Ce nouvel AST typé a exactement la même structure que l'AST pris en entrée, et contient juste des informations supplémentaires : chaque expression (et chaque sous-expression) est annotée par le type qui a été calculé par le vérificateur.
3. Le traducteur de Mini-C vers PIMP à proprement parler prend en entrée l'AST Mini-C typé.

Autres formes de types définies par l'utilisateur

On peut également intégrer à Mini-C d'autres formes courantes de types définis par l'utilisateur. Une *énumération* est un type défini par un ensemble fini de symboles. On trouve le concept aussi bien en C avec le mot-clé `enum`

```
enum tete {
  valet; dame; roi;
}
```

qu'en caml avec un type algébrique où tous les constructeurs sont constants.

```
type tete =
  Valet | Dame | Roi
```

Pour traduire un programme utilisant des énumérations vers un langage de plus bas niveau comme PIMP, on numérote les symboles de chaque énumération.

$$\begin{aligned} \texttt{valet} &\mapsto 0 \\ \texttt{dame} &\mapsto 1 \\ \texttt{roi} &\mapsto 2 \end{aligned}$$

Il suffit alors de traduire dans le programme chaque occurrence d'un tel symbole par son numéro.

Un peu plus riche, un type *union* décrit pour un type de donnée une alternative entre plusieurs formats. En C par exemple, on déclare avec

```
union s {
  point p;
  int  n;
};
```

un type `s` désignant une donnée qui peut être soit un point soit un entier. On trouve un concept similaire en caml avec la définition de type algébrique

```
type s =
  | P of point
  | N of int
```

Nuance entre ces deux versions : en caml on introduit impérativement des *constructeurs*, ici les symboles `P` et `N`, qui permettent de distinguer les différents formats. L'opération de filtrage permet alors, partant d'une donnée concrète de type `s` en caml, de tester sa forme. En C en revanche, la représentation d'une donnée de type `s` ne permet pas par défaut de distinguer si à l'intérieur se trouve un point ou un entier. Il faut ajouter explicitement ces informations dans la structure, par exemple à l'aide d'une énumération des différents formes possibles.

Compilateur Mini-C, bilan

On peut obtenir un compilateur complet de Mini-C (langage impératif avec fonctions et structures de données) vers l'assembleur MIPS en combinant les éléments suivants du cours :

Élément	Référence	Langage utilisé
analyseur lexical	chapitre 3 et DM	ocamllex
analyseur grammatical	chapitre 4 et DM	menhir
vérificateur de types	chapitre 5 et DM	caml
traducteur Mini-C vers PIMP	section 6.11	caml
allocateur mémoire	section 6.10	*PIMP*
traducteur PIMP vers LLIR	sections 6.4 et 6.9	caml
traducteur LLIR vers MIPS	section 6.8	caml

On pourrait ensuite continuer dans plusieurs directions, et notamment enrichir le langage source et étendre le compilateur pour traiter de nouvelles fonctionnalités de plus haut niveau (objets ? exceptions ? programmation fonctionnelle ?), ou améliorer le compilateur lui-même pour qu'il produise du code assembleur optimisé dont l'exécution sera (beaucoup) plus efficace.

7 Et ensuite

Revenons sur ce que nous avons parcouru pendant ce semestre sur la compilation et la théorie des langages de programmation, et en particulier sur :

— ce qu'il faut en retenir, même pour ceux qui ne voudraient plus jamais toucher au domaine,
— ce qui reste, pour ceux qui voudraient aller plus loin.

Définition d'un langage de programmation

Un langage de programmation est défini par *des* syntaxes et *des* sémantiques. Côté syntaxe :

— la syntaxe concrète définit ce que l'on écrit lorsque l'on programme, et attise parfois les passions et les trolls,
— la syntaxe abstraite définit les structures qui sont représentées par un texte concret, et est également ce sur quoi on peut raisonner [4].

Côté sémantique :

— la sémantique statique, en particulier avec les types, distingue les programmes qui ont du sens de ceux qui n'en ont pas,

4. De là à dire que les trolls résonnent plus qu'ils ne raisonnent, il n'y a qu'un pas.

— la sémantique dynamique décrit les comportements à attendre d'un programme, et spécifie notamment les résultats et les effets.

Exécution d'un programme

Pour permettre l'exécution d'un programme donné dans un langage source, l'interprétation et la compilation sont deux voies de natures différentes. On peut observer cette différence par les entrées et sorties produites dans chaque cas.

— Interpréter un programme, c'est exécuter son comportement. Dans ce contexte on a :
  — en entrée : un programme *et* des entrées,
  — en sortie : le résultat.
— Compiler un programme, c'est le traduire vers un autre langage (qui pourra ensuite être exécuté à l'aide de ses mécanismes propres). Dans ce contexte on a :
  — en entrée : un programme,
  — en sortie : un programme *équivalent*.

Au-delà de leurs différences fondamentales, ces deux voies ont de nombreux aspects techniques communs. Elles mettent en œuvre l'analyse d'un texte source (analyse lexicale, analyse grammaticale), la manipulation d'une représentation symbolique du programme (AST), et imposent le respect d'une sémantique (correction).

Un domaine tentaculaire

Étudier la compilation, c'est approfondir sa culture et sa compréhension des langages de programmation. On y décortique les langages et leurs mécanismes. On en ressort meilleur programmeur, mieux apte à écrire des programmes sûrs et efficaces, et on améliore sa capacité à comprendre et corriger les erreurs dans ses programmes.

C'est aussi une occasion d'explorer et de connecter des champs multiples de l'informatique :

— l'algorithmique, avec des algorithmes avancés d'analyse et d'optimisation,
— l'informatique théorique, avec la sémantique, les langages formels, la calculabilité,
— la technologie informatique, lors de l'interface avec l'architecture et le système,
— la programmation elle-même : un compilateur est un programme d'envergure, dont l'écriture regroupe tous les aspects précédents.

Pour aller plus loin

Certains des chapitres de ce cours ouvrent sur des domaines que vous pourrez explorer dans la suite de votre cursus.

À la suite des chapitres 3 et 4 s'étendent les domaines de la *calculabilité* et de la *complexité*, qui donnent les *fondements théoriques de l'algorithmique*. On y cherche notamment à caractériser les dispositifs de calcul (automates, machines de Turing, machines RAM, etc) et les problèmes qu'ils peuvent résoudre. On y découvre avec la notion d'indécidabilité que certains problèmes ne peuvent pas être résolus par des algorithmes. On y caractérise aussi certains problèmes algorithmiques qui, bien que solubles, sont intrinséquement difficiles. On ne sait en revanche toujours pas si P est égal ou non à NP.

À la suite des chapitres 2 et 5, on arrive à la *science des langages de programmation*. Outre la modélisation du comportement des programmes, on y vise à créer de nouveaux outils d'analyse des programmes et de leurs comportements, de nouveaux systèmes de types et de nouveaux langages de programmation permettant d'améliorer la sûreté des programmes futurs. C'est ici que l'on traite de preuve de programmes. C'est de là que viennent les assistants à la preuve, et les plus intéressants des nouveaux langages de programmation. À ce propos, connaissez-vous Rust ?

Le chapitre 6 est le début de l'*implémentation des langages de programmation*. On y veut faire le lien entre des mécanismes de programmation de haut niveau, agréables au programmeur, et les opérations plus élémentaires accessibles à la machine. On y cherche également à ce que l'exécution des programmes soit ensuite la plus efficace possible. Cela peut passer par des analyses fines de la structure des programmes et de leurs opérations pour détecter des endroits où des optimisations sont possibles, ou encore par la meilleure utilisation des capacités matérielles de la machine cible.