École nationale supérieure d'informatique pour l'industrie et l'entreprise

Cours d'assembleur et de compilation

Semestre 3

15 décembre 2021

Guillaume Burel

Table des matières

1. Introduction

Un ordinateur, plus particulièrement son processeur, comprend les informations sous forme de signal électrique discret. En général on note par 1 la présence d'une tension et par 0 son absence, ce qui représente un bit d'information. Le signal électrique fournit donc une séquence de bits. Pour fonctionner, il faut que ce signal électrique respecte le format attendu par le processeur. Pour chaque (famille de) processeur, il existe une spécification des instructions à exécuter en fonction des séquences de bits reçues en entrée : on parle de code machine ou de langage machine.

À l'aube de l'informatique, ces séquences de bits étaient explicitement écrites par le programmeur : instructions et données à traiter étaient codées en binaire pour être exécutées par la machine. Pour cela, on pouvait par exemple utiliser des cartes perforées : un trou représente un bit de valeur 1, l'absence de trou un bit de valeur 0.

Il s'est vite avéré que le travail d'écrire du code machine directement était fastidieux et prompt à causer des erreurs. En effet, pour un cerveau humain, il est assez difficile de retenir des codes binaires qui représentent les opérations à effectuer, alors qu'il est plus facile de manipuler des noms. Il est ainsi plus facile de savoir que SUB correspond à l'instruction pour effectuer une soustraction plutôt que le code binaire 010110. On utilise alors des mnémoniques, c'est-à-dire des noms, qui permettent de facilement se rappeler et manipuler les codes des instructions, les registres, les adresses, etc.

Définition 1.1 (Langage assembleur)**.** *Un langage assembleur est une représentation* lisible *du langage machine où les instructions, les adresses et les données peuvent être abstraites à l'aide de noms.*

Le processeur n'est par contre pas capable d'exécuter un programme en langage assembleur directement : il ne peut comprendre que le code machine. Il faut donc traduire le programme en langage assembleur en code machine. Dès 1949, on a vu apparaître l'utilisation de noms pour désigner de façon facilement compréhensible les instructions. Néanmoins, la traduction vers le code machine se faisait alors à la main, ce qui, outre le fait que cette étape était longue et fastidieuse, pouvait entraîner des erreurs de traduction. Assez rapidement, cette tâche de traduction a donc été dévolue à un programme informatique : le programme assembleur.

Définition 1.2 (Programme assembleur)**.** *Un programme assembleur est un exécutable qui convertit un programme écrit en langage assembleur en du code machine.*

Le premier programme assembleur a été écrit en 1954 pour l'IBM 701.

En général on désigne par « assembleur » seul aussi bien le langage assembleur que le programme assembleur, en fonction du contexte.

On remarquera que même écrit en langage assembleur, un programme a besoin d'être codé en binaire pour pouvoir être stocké, lu par la machine, etc. Les cartes perforées ont été largement utilisées jusque dans les années 1970 qui ont vu la démocratisation des supports de stockage magnétiques.

Contrairement à d'autres langages de programmation de plus haut niveau, le langage assembleur reste proche du langage machine. On peut même parfois lire qu'il existe une bijection entre les deux, ce qui n'est pas tout à fait exact. En effet, il est possible de renommer des variables, ou d'utiliser différentes façons pour indiquer des adresses, sans que le code machine final ne soit différent. À partir d'un code machine, on peut obtenir un programme en langage assembleur lui correspondant : on parle de désassemblage. Néanmoins, cette opération de désassemblage n'est pas unique : on peut faire correspondre plusieurs programmes en langage assembleur au même code machine.

Une des limites principales des langages assembleurs est qu'ils sont spécifiques à une machine (ou à une famille de machines) donnée. Dès les années 50 est apparu le besoin de langages de plus haut niveau que l'assembleur, de façon à abstraire certaines particularités propres à la machine (registres en nombre finis, instructions de branchement basiques, etc.) pour mieux se concentrer sur les aspects algorithmiques et pouvoir passer à l'échelle sur des projets plus complexes. Un des objectif était également de pouvoir écrire un seul programme pour des architectures différentes. Pour pouvoir utiliser ces langages, il faut alors soit disposer d'un interpréteur, c'est-à-dire un exécutable qui va lire le programme et évaluer son contenu ; soit traduire le programme en code machine exécutable : on parle alors de compilation.

Ces langages se sont heurtés au scepticisme des programmeurs de l'époque qui avaient l'habitude de produire des codes assembleurs très efficaces et qui pensaient qu'il ne serait pas possible de produire automatiquement des exécutables aussi efficaces à partir de langages de haut niveau. Un bon compilateur ne peut donc se contenter de traduire naïvement le langage de haut niveau, il doit l'*optimiser*. Ces optimisations, associées au gain d'échelle offert par l'abstraction, ont permis la généralisation de l'utilisation de langages de haut niveau.

Un compilateur ne traduit pas forcément un langage en code machine, il peut produire du code dans un langage intermédiaire qui sera ensuite lui-même compilé (par exemple, le langage intermédiaire peut être du C) ou interprété (par exemple, du bytecode).

Définition 1.3. *Un compilateur est un exécutable qui traduit un langage de haut niveau vers un langage de plus bas niveau.*

La qualification de haut ou bas niveau pour un langage est subjective. Ainsi, C est un langage de haut niveau si on le compare à de l'assembleur, mais les compilateurs pour certains langages produisent du C (l'avantage étant qu'il existe ensuite des compilateurs de C vers de nombreuses architectures, ce qui évite de devoir écrire un compilateur pour chacune d'elle). Par la suite, on se contentera de parler de langage source et de langage cible.

Exemple 1.1 *:* Le premier compilateur optimisant, écrit en 1957, traduisait du Fortran en code machine pour l'IBM 704. Les compilateurs de Fortran sont toujours parmi les meilleurs à l'heure actuelle en terme d'optimisation. Ceci s'explique par la relative simplicité du langage, mais aussi par l'utilisation de Fortran pour le calcul scientifique qui a engendré le besoin d'obtenir du code très efficace.

Les compilateurs pour C produisent en général du code machine (exemple : gcc). Le compilateur java de Sun produit du bytecode qui est ensuite interprété par une machine virtuelle, la JVM. Ocaml dispose de deux compilateur, `ocamlc` et `ocamlopt`, produisant respectivement du bytecode et du code machine.

Un exécutable qui génère du PDF à partir d'un autre langage (exemple : LaTeX, SVG, PostScript, etc.) est aussi un compilateur.

Un préprocesseur peut également être vu comme un compilateur du langage avec macros vers le langage pur.

Lex et Yacc sont aussi des compilateurs : ils traduisent des expressions régulières et des grammaires hors-contexte vers du code C. (Cf. l'acronyme de Yacc : *Yet Another Compiler Compiler).*

Dans la plupart des exemples de ce cours, on considérera comme langage source un sous-ensemble du langage Pascal qu'on appellera Pseudo-Pascal. Sa syntaxe est fournie en annexe A.4. Le langage cible sera quant à lui le langage assembleur RISC-V dont on donne en annexe A.3 les instructions les plus courantes.

En général, un compilateur ne se contente pas de traduire un langage dans un autre, il est capable de signaler des erreurs de syntaxe, de sémantique (par exemple via une vérification de type) si possible de façon compréhensible par l'utilisateur, il fait des optimisations qui peuvent viser plusieurs objectifs parfois contradictoires : vitesse d'exécution, taille du code, utilisation de la mémoire (notamment pour les applications embarquées), etc.

Première partie

Assembleur

2. Langage machine

Le langage assembleur est une abstraction du langage binaire directement compréhensible par le processeur, le langage machine. Hors exception, les processeurs traitent successivement des instructions qui leur sont spécifiques.

2.1. Jeux d'instructions

Définition 2.1 (Instruction)**.** *Une instruction machine est une opération élémentaire (atomique) qu'un processeur peut effectuer.*

Exemple 2.1 *:* En général, un processeur possède des instructions pour faire :
- des opérations arithmétiques ; celles-ci peuvent agir sur des entiers, ou sur des nombres à virgule flottante, il y a bien des instructions distinctes suivant le type de nombres (autrement dit, l'instruction d'addition sur les entiers est différente de l'instruction d'addition sur les nombres à virgule flottante) ;
- des opérations logiques ("et", "ou" mais aussi rotation, décalage, etc.)
- des instructions de transfert, qui permettent de copier des données entre la mémoire vive de l'ordinateur et les registres ou la pile du processeur ;
- des branchements, qui permettent de se déplacer dans le programme, et qui peuvent être conditionnels ou non.

Définition 2.2 (Jeu d'instructions)**.** *Un jeu d'instructions est l'ensemble des instructions qu'un processeur peut réaliser.*

À l'origine, chaque nouvelle machine définissait son propre jeu d'instruction. Néanmoins, assez rapidement, on a cherché à pouvoir réutiliser les jeux d'instructions des anciens modèles, de façon à améliorer la réutilisabilité du code existant, de pouvoir le porter plus facilement entre différentes machines, et d'avoir des processeurs qui soient rétrocompatibles avec les générations précédentes. Par conséquent, certains jeux d'instructions sont communs à plusieurs processeurs, on parle alors de famille de processeurs.

Exemple 2.2 *:* Parmi les familles les plus courantes de processeurs, on peut citer :

x86 Cette famille de processeurs est basé sur le jeu d'instruction du processeur Intel 8086. Ce jeu d'instruction a été petit à petit étendu pour ajouter de nouvelles fonctionnalités en dur dans le processeur ; par exemple, les processeurs actuels de cette famille possèdent en général des extensions SSE (qui permettent de travailler sur plusieurs données à la fois, ce qui est utile dans des applications de traitement du signal ou graphiques), ou encore l'extension x86-64 qui permet de travailler sur des données de 64bits.

PowerPC ce jeu d'instruction est surtout connu pour avoir été utilisé dans les processeurs équipant les ordinateur de la marque Apple jusqu'en 2006 (quand Apple est passé à la famille x86). Il est aujourd'hui encore utilisé dans des systèmes embarqués (par exemple sur le rover Curiosity actuellement en activité sur Mars), ou dans des consoles de jeu (Wii, Playstation 3, ...)

ARM ce jeu d'instruction est particulièrement adapté pour les systèmes embarqués et mobiles, par exemple les smartphones. C'est par conséquent le jeu d'instructions le plus utilisé et celui dont la plus grande quantité de processeurs est produite.

MIPS bien que conçu pour être généraliste, ce jeu d'instruction a surtout été utilisé dans des systèmes embarqués et des consoles de jeu (Playstation 1 et 2, Nintendo 64).

RISC-V ce jeu d'instruction a été développé par l'université de Californie à Berkeley en 2010 dans le but de fournir un standard ouvert servant de base à la conception de processeurs. Il n'y a en particulier pas de frais à payer pour son implémentation. Bien que développé initialement dans le secteur académique, avec une volonté de fédérer les différentes avancées dans un standard unique pour faire progresser plus rapidement la recherche sur les jeux d'instructions, il y a toujours eu l'idée de faire passer les bons choix de conception résultants dans l'industrie. Ce standard est de plus en plus adopté par cette dernière : outre de petites entreprises qui se sont construites autour de ce standard, comme SiFive, des grands groupes comme NVidia, Apple ou Huawei commencent à s'y intéresser de près. (Notamment, concernant Huawei, suite à l'interdiction américaine de l'utilisation d'Intel ou d'ARM.) Du fait de sa simplicité et de son potentiel, c'est principalement ce jeu que nous étudierons par la suite.

2.2. Classification des familles de processeurs

2.2.1. CISC vs. RISC

Lors de la conception d'un processeur, on peut raisonnablement se poser la question suivante : vaut-il mieux avoir beaucoup d'instructions très spécifiques pour pouvoir répondre au besoin du code ? ou vaut-il mieux un petit ensemble d'instructions de base ? On peut alors distinguer deux classes de familles de processeurs :

CISC *Complex Instruction Set Computer* Il s'agit de jeux d'instructions dans lesquels une instruction atomique peut réaliser plusieurs opérations à la fois, par exemple un chargement d'une donnée depuis la mémoire et une opération arithmétique sur cette donnée, ou bien un calcul sur plus de deux données à la fois.

Un exemple typique de jeu d'instruction CISC est le x86.

RISC *Reduced Instruction Set Computer* Il s'agit de jeux d'instructions relativement petits (en nombre d'instructions) avec des instructions simples et relativement générales.

Par les jeux d'instructions RISC on trouve ARM, MIPS et PowerPC, et bien entendu RISC-V.

Parmi les avantages du CISC, on trouve :

— le code obtenu peut être plus compact puisqu'on peut utiliser une seule instruction pour effectuer plusieurs opérations (on obtient typiquement un facteur $\frac{1}{2}$) ;
— il est possible de fortement optimiser un code, en choisissant les instructions les plus adaptés aux calculs demandés.

Parmi ses inconvénients, on peut citer :

— certaines des instructions complexes durent en fait plus d'un cycle d'horloge ; par exemple, l'instruction pour la division dans le Motorola 68000 a besoin de 160 cycles d'horloges ! On peut donc obtenir du code plus compact en taille, mais pas forcément plus rapide ;
— les compilateurs ne produisent pas forcément les instructions les plus optimales pour un calcul donné ; c'est un problème assez complexe.

Le CISC a comme avantage d'être plus facile à concevoir, avec des processeurs contenant moins de transistors, ce qui fait qu'ils coûtent moins chers, consomme moins de courant et produisent par conséquent moins de chaleur. Cela explique leur quasi-omniprésence dans le monde embarqué et mobile. De plus, on peut compter sur les compilateurs pour essayer d'optimiser le code et avoir un nombre d'instructions finales assez réduit.

2.2.2. Accès aux données

On peut aussi classifier les jeux d'instructions en fonction de la façon dont ils accèdent aux données, comment ils les chargent et les stockent. Nous allons illustrer ceci sur la traduction d'une addition `C := A + B`.

"0 adresse" ce type d'instructions utilise une pile dans laquelle sont stockées toutes les données. L'addition est alors traduite par

```
PUSH A
PUSH B
ADD
POP C
```

"à accumulateur" ce type d'instructions utilise un seul registre fixe dans lequel sont stockés tous les calculs. Cela donne

```
LOAD A
ADD B
STORE C
```

"à registre-registre" ce type d'instruction utilise plusieurs registres à donner en paramètre. Une valeur peut donc être chargée puis réutilisée plusieurs fois.

```
LOAD R0, A
LOAD R1, B
```

```
ADD R2, R0, R1
STORE R2, C
```

“à registre-mémoire” ce type d’instruction utilise à la fois des registres et des adresses mémoire.

```
LOAD R0, A
ADD R1, R0, B
STORE R1, C
```

Cela permet d’utiliser moins de registre, mais cela va à l’encontre de l’approche RISC.

Un jeu d’instruction peut en général avoir des instructions de différents types d’accès aux données.

À noter : x86 est à la fois accumulateur et à registre : la plupart des instructions mettent leur résultat dans le registre AX.

3. Langage assembleur

Le langage assembleur est une abstraction du langage machine qui permet l'utilisation de noms plus facile à retenir pour les instructions, les registres, les adresses, etc.

3.1. Différence entre langage machine et langage assembleur

Le langage machine est un langage binaire, tandis que le langage assembleur utilise des noms (identifiants). Néanmoins, leur différence va au-delà. On peut citer les distinctions suivantes :

Abstraction des opérateurs Les instructions sont identifiées à l'aide de noms facilement mémorisables au lieu de code binaire.

Ainsi, on utilise le mot `sub` et non pas le code `011010`.

Abstraction des noms des registres De façon à mieux refléter leur usage conventionnel, les registres sont identifiés par une abréviation plutôt que par leur numéro.

Ainsi, on peut parler du registre `sp` qui contiendra l'adresse du sommet de la pile d'appel (*stack pointer*) plutôt que du registre `x2`.

Étiquettes symboliques pour les adresses mémoires Les adresses mémoires, en particulier celles correspondant à des points du code, ou bien celles correspondant à des données, peuvent être identifiées à l'aide d'une étiquette. Ainsi, on peut utiliser l'étiquette pour facilement désigner le point du code ou la donnée.

Cela est particulièrement utile quand on modifie le programme et que l'adresse réelle est modifiée (parce que la position dans le programme a changé), mais que l'étiquette elle ne change pas.

Par exemple, on peut donner une étiquette à l'instruction au début d'une boucle, et faire un branchement vers cette étiquette à la fin de la boucle.

Directives Les directives permettent d'indiquer au programme assembleur comment traduire la suite du programme, mais elles ne génèrent pas de code en tant que tel. Par exemple, on peut avoir des directives pour indiquer quelle partie du programme correspond à des données et quelle partie à des instructions. On peut aussi avoir des directives indiquant comment présenter les données.

Par exemple en RISC-V, la directive `.asciiz` permet d'écrire une chaîne de caractères entre guillemets plutôt que de donner la suite de code ASCII terminée par 0 correspondant, par exemple `.asciiz "This is a string."`

Commentaires En général il est possible d'insérer des commentaires, ce qui est particulièrement utile en cas de programme écrit à la main.

Pseudo-instructions Il s'agit d'instructions qui sont fournies par le langage assembleur, mais qui ne correspondent pas à une vraie instruction en langage machine. Ces pseudo-instructions sont ensuite traduites en une ou plusieurs vraies instructions machine.

Par exemple en RISC-V, il est possible d'utiliser la pseudo-instruction `ret`, qui retourne à la fonction appelante. Cette pseudo-instruction est traduite par `jalr zero, 0(ra)` : on saute à l'instruction à l'adresse contenue dans `ra`, et on ne garde pas l'instruction de l'adresse qui suit `ret` puisqu'on la met dans `zero` qui vaut toujours 0. De même, la pseudo-instruction `mv rd, rs` est en fait `addi rd, rs, 0`.

Macros Certains langages assembleurs (mais pas tous) permettent de définir des macros, qui permettent d'éviter la duplication de code. Exemple :

```
.macro maFonction(arg)
  add a0, arg, arg
  addi a0, a0, 1
```

Attention, ici ni `arg` ni `a0` ne sont des variables locales ! En particulier il peut s'agir du même registre.

3.2. Programme assembleur

Pour gommer les différences entre le langage assembleur et le code machine, le programme assembleur doit donc effectuer les tâches suivantes :

— remplacer les pseudo-instructions par la ou les instructions équivalentes ;
— remplacer les noms de registres par leur numéro ;
— remplacer les adresses symboliques par les véritables adresses (suivant les instructions, décalage par rapport à l'instruction courante ou adresse absolue) ;
— remplacer les instructions assembleur par leur équivalent machine en binaire.

Le programme assembleur produit alors un fichier objet (extension `.o` en général) qui contient, en plus du code machine et des données statiques, des informations sur l'utilisation du code, par exemple les étiquettes à exporter qui indiquent quelles fonctions peuvent être utilisées dans le code.

On utilise ensuite un éditeur de lien pour lier les différents fichiers objet entre eux et pour générer l'exécutable.

4. Le langage assembleur RISC-V

RISC-V est un standard ouvert de jeux d'instructions. Il comporte un nombre relativement réduit d'instructions simples, et utilise un nombre élevé de registres qui sont quasiment tous interchangeables.

RISC-V est décliné en plusieurs versions en fonction du nombre de bits sur lequel il se base : 32, 64 ou 128. C'est un standard modulaire : un certain nombre d'instructions dites de base sont définies, et celles-ci peuvent être complétées par une ou plusieurs extensions. Une implémentation de RISC-V (autrement dit, un processeur) n'est pas obligé d'implémenter toutes les extensions mais uniquement celles qui l'intéressent. Cela permet d'avoir des conceptions plus simples pour des processeurs qui n'ont pas besoin d'être génériques, par exemple pour de simples contrôleurs. Les versions de RISC-V sont alors décrites par des noms de la forme : RV$nABCD$ où RV signifie RISC-V, n est le nombre de bits, et $ABCD$ sont la base et les extensions utilisées. On a par exemple RV32IM : 32 bits avec base sur les entiers et multiplications.

Les bases et extensions suivantes font partie du standard :

I base sur les entiers (Integer) ; définit la plupart des opérations courantes sur les entiers (addition, décalage, et bit à bit, etc) mais pas la multiplication ni la division ; ainsi que des instructions de saut, de branchement, de lecture/écriture en mémoire et d'appel système ;

E base pour les systèmes embarqués ; essentiellement identique à I, mais il n'y a que 16 registres disponibles et non 32 ;

M extension sur la multiplication ; ajoute des opérations de multiplication, de division et de reste sur les entiers ;

C extension avec instructions compressées ; ajoute des instructions qui tiennent sur 16 bits au lieu de 32, ce qui permet d'avoir un code deux fois plus compact ; utile par exemple dans du code embarqué ;

F extension sur les nombres à virgule flottante simple précision ; ajoute des instructions conformes au standard arithmétique IEEE 754-2008 sur les flottants sur 32 bits ;

D extension sur les nombres à virgule flottante double précision ; ajoute des instructions conformes au standard arithmétique IEEE 754-2008 sur les flottants sur 64 bits ; nécessite l'extension F ;

G raccourci pour IMFDAZicsr_Zifence : instructions présentes dans un processeur polyvalent et générique, tel qu'on peut en trouver dans un PC.

On trouvera en annexe A.3 les instructions les plus courantes de la version RV32IM avec laquelle nous allons travailler.

Deuxième partie

Compilation

5. Architecture générale d'un compilateur

Correction

Pour qu'un compilateur soit correct, il faut que le code produit ait le même comportement que celui attendu pour le programme source. Pour cela, il est nécessaire de connaître la *sémantique* des langages source et cible. Dans le cas d'un langage machine, cette sémantique est définie par le fonctionnement de la machine elle-même. Dans les autres cas, la sémantique a besoin d'être spécifiée. Une façon de spécifier la sémantique est de donner un ensemble de règles d'inférence qui décrivent comment évolue l'environnement et quels sont les résultats des calculs. On trouvera en annexe A.5 une fiche décrivant la sémantique de Pseudo-Pascal.

Architecture

Les langages sources tels que Pseudo Pascal diffèrent des langages cibles comme RISC-V sur de nombreux points :

Pseudo Pascal	RISC-V
opérateurs ordinaires	opérateurs ad hoc (`+k`, `<<k`)
expressions structurées	instructions élémentaires
instructions structurées	branchements
pile implicite	pile explicite
variables en nombre illimité	registres en nombre fini

Passer d'un programme Pseudo Pascal en un programme RISC-V en un seul jet est virtuellement impossible. Par conséquent, on passe par de nombreuses étapes intermédiaires, en ne changeant qu'un petit aspect à chaque fois.

À chaque étape, on dispose d'un langage intermédiaire (on parle aussi de *représentation intermédiaire*) qui ne diffère du précédent qu'en un petit nombre de points.

Chaque langage intermédiaire dispose de sa propre syntaxe abstraite et (en principe) de sa propre sémantique. La spécification de chaque phase est donc limpide : étant donné un programme exprimé dans le langage intermédiaire L_k , elle produit un programme exprime dans le langage intermédiaire L_{k+1} dont la sémantique est équivalente.

Il peut arriver que les différentes phases partagent certaines données, par exemple un table des symboles globale. Néanmoins, on pourra supposer par la suite que ce n'est pas le cas. Chaque phase est alors une fonction pure.

Pour des raisons historiques, on distingue trois types de phases (*front-end, middle-end, back-end*).

Front-end

La première tâche du compilateur est de comprendre l'expression du langage source. Cela se fait en plusieurs étapes :

1. Analyse syntaxique : permet de vérifier que l'expression est bien une phrase du langage source syntaxiquement correcte, on dit aussi qu'elle est bien formée. Cela nécessite donc une définition formelle du langage source. Exemple en français : « Le lion mange de la viande » est syntaxiquement correcte et « le lion viande » n'est pas syntaxiquement correcte. En pratique, l'analyse cette phase est divisée en deux traitements : l'analyse lexicale ou scanning (repérer les césures de mots, la ponctuation) et l'analyse syntaxique ou parsing (vérifier les règles de grammaire pour l'exemple du français).

2. Analyse sémantique : permet de vérifier que l'expression a un sens dans le langage source (on peut dire aussi analyse sensible au contexte, context sensitive analysis, CSC en anglais). Cela nécessite une sémantique précise pour le langage source. Exemple en français : « le lion dort de la viande » est syntaxiquement correcte (sujet, verbe, complément d'objet) mais n'a pas de sens défini. Ici, on peut être amené à se demander si les variables w ; x ; y et z ont été déclarées, si elles ont été initialisées, si les types sont correctement utilisés, etc.

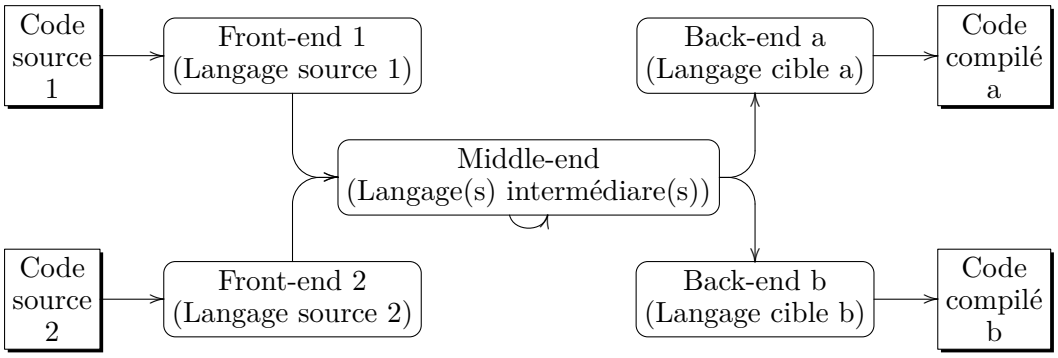
Ces traitements sont regroupé dans ce qui est appelé le front-end du compilateur. Ces deux traitements sont largement automatisés aujourd'hui grâce à l'application des résultats de la théorie des langages. On dispose ainsi d'outils permettant de générer les analyseurs lexicaux et syntaxiques à partir de la description du langage source sous forme de grammaire (cf. section 7).

Back-end

C'est lors de cette phase qu'est généré le code pour le langage cible. Pour obtenir un compilateur d'un même langage vers des architectures différentes, seuls les back-ends ont besoin d'être spécifiques à l'architecture cible, ce qui permet de réutiliser le reste du compilateur.

Middle-end

Ce sont des phases qui permettent d'ajouter des optimisations. Si ces optimisations travaillent sur les mêmes langages intermédiaires (c'est-à-dire si $L_k = L_{k+1}$), elles peuvent alors être optionnelles (cf. l'option -O de gcc). L'ajout de cette phase a permis de commencer à parler de compilateur optimisant.



En pratique, des optimisations sont effectuées à tout moment pendant la compilation. Certaines peuvent avoir lieu lors de la traduction de l'arbre de syntaxe abstrait vers la première représentation intermédiaire, d'autres ne sont possibles qu'en utilisant les spécificités du langage cible et n'interviennent donc que dans le back-end. Il n'est donc pas toujours possible de distinguer clairement les trois types de passes d'un compilateur.

L'un des avantages de séparer le compilateur en de nombreuses passes est de pouvoir les réutiliser. Nous avons déjà mentionné le cas des différentes architectures cibles, il est également possible de réutiliser certaines optimisations, ou encore l'analyse syntaxique, etc. Il existe des librairies proposant des optimisations et des générateurs de code. On citera en particulier le projet LLVM (`http://llvm.org/`).

Dans ce cours, on étudiera les passes suivantes : analyse lexicale → analyse syntaxique → analyse sémantique → sélection d'instruction → création du graphe de flot de contrôle → explicitation des conventions d'appel → allocation de registres.

6. Sémantique

Pour définir un langage de programmation, il ne suffit pas de décrire sa syntaxe, c'est-à-dire l'ensemble des expressions valides dans ce langage. Il faut également décrire sa sémantique, c'est-à-dire comment ces expressions seront interprétées. Dans le cas d'un langage machine, la sémantique est clairement définie par le comportement physique de la machine. Toutefois, cette sémantique est en général documentée de façon plus accessible à un être humain, avec la description des modifications des registres en fonctions des instructions. Dans le cas d'un langage de plus haut niveau, il existe plusieurs méthodes pour définir sa sémantique. La plus simple est appelée sémantique opérationnelle, elle consiste en un système de transition qui décrit comment l'état de la machine (modélisé de façon abstraite) évolue en fonction du programme.

6.1. Sémantique opérationnelle de Pseudo Pascal

Comme on cherche à décrire la sémantique d'un langage de haut niveau, il serait absurde de la définir par le comportement d'une machine réelle. Par conséquent, on commence par abstraire le fonctionnement d'une telle machine. On considérera donc un ensemble d'états, qui représenteront les états physiques de la machine de façon abstraite. Dans le cas de Pseudo Pascal, on considérera des états constitués :

— d'un environnement global G, qui associe aux variables globales des valeurs ;
— d'un environnement local E, qui associe aux variables locales des valeurs ;
— d'un tas H, qui associe à des adresses des suites finies de valeurs (pour le contenu des tableaux).

Remarque 6.1 *:* On peut utiliser d'autres méthodes pour modéliser l'emplacement mémoire des tableaux, certaines étant plus précises que d'autres (c'est-à-dire correspondant plus à ce qui se passe en machine), ce qui permet de décrire le fonctionnement de plus de programmes, par exemple en cas de dépassement de tableau (en C, dans le programme

```
struct {int t[3]; int k;} e;
e.t[3] = 4;
```

la valeur de `e.k` est modifiée). On parle de différents modèles mémoire.

On va ensuite décrire comment un programme modifie les états. Pour cela, on va considérer différents type de jugements :

— $G, H, E/i \to G', H', E'$: dans l'état G, H, E, l'évaluation de l'instruction i termine et mène à l'état G', H', E' ;

— $G, H, E/e \to G', H', E'/v$: dans l'état G, H, E, l'évaluation de l'expression e produit la valeur v et mène à l'état G', H', E' ;
— $G, H, E/c \to G', H', E'/b$: dans l'état G, H, E, l'évaluation de la condition c produit le booléen b et mène à l'état G', H', E' ;
— $p \to$: le programme p s'exécute sans erreur et termine.

Les jugements sont définis à l'aide de règles d'inférence, c'est-à-dire qu'ils sont définis par induction (cf. cours de logique). L'ensemble des règles d'inférences est donné dans l'annexe A.5 Nous n'en détaillons ici que certaines. Par exemple, l'évaluation d'une constante k est donné par la règle

$$\frac{}{G, H, E/k \to G, H, E/k}$$

La constante k est évaluée en k [1] ce qui ne change pas l'environnement.

L'évaluation de l'addition est donnée par la règle

$$\frac{G, H, E/e_1 \to G', H', E'/v_1 \qquad G', H', E'/e_2 \to G'', H'', E''/v_2}{G, H, E/e_1 + e_2 \to G'', H'', E''/\underline{v_1 + v_2}}$$

où $\underline{v_1 + v_2}$ désigne le résultat de l'addition des valeurs v_1 et v_2. On remarque que e_1 est évalué avant e_2, ce qui se traduit par le fait que l'environnement G, H, E est d'abord transformé en l'environnement G', H', E' par l'évaluation de e_1, puis que c'est cet environnement qui est utilisé dans l'évaluation de e_2. La définition de la sémantique décrit donc bien l'ordre d'évaluation des expressions. Les règles "Évaluation des arguments d'une fonction" et "Appel d'une fonction définie" montre par ailleurs que les arguments passés à une fonction sont d'abord évalués (de gauche à droite) avant que la fonction ne le soit, on parle d'appel par valeur.

Concernant la sémantique des conditions, on notera le caractère coupe-circuit des opérateurs booléens : si c_1 s'évalue en $false$, la condition c_1 `and` c_2 sera évaluée en $false$ sans que c_2 ne soit évaluée.

En ce qui concerne la sémantique des instructions, on notera les règles pour la conditionnelle : on évalue d'abord la condition, puis suivant sa valeur on évalue uniquement une des deux branches ; et pour la boucle : on évalue la condition, si elle est fausse on ne fait rien sinon on évalue la séquence du corps de la boucle suivi de la boucle. On aurait pu remplacer la règle "Boucle (si)" par la règle équivalente suivante :

$$\frac{\begin{array}{cc} G, H, E/c \to G', H', E'/true & \\ G', H', E'/i \to G'', H'', E'' & G'', H'', E''/\texttt{while } c \texttt{ do } i \to G''', H''', E''' \end{array}}{G, H, E/\texttt{while } c \texttt{ do } i \to G''', H''', E'''}$$

Enfin, dans la règle donnant la sémantique d'un programme (jugement de type $p \to$), on voit que les variables globales sont automatiquement associées à leur valeur par défaut (contrairement à ce qui se passe par exemple en C).

Exemple 6.1 *:* On cherche à évaluer la sémantique du programme suivant :

1. En réalité, il faudrait distinguer le k constante du langage du k valeur évaluée.

```
var t : array of integer
swap(integer x, integer y)
  var tmp : integer
  begin
    tmp := x;
    x := y;
    y := tmp
  end
begin
  t := new array of integer [2];
  t[0] := 1;
  swap(t[0],t[1])
end
```

On part d'un état $\{\texttt{t} \mapsto \mathit{nil}\}, \emptyset, \emptyset$ dans lequel on doit évaluer le corps du programme (`begin t := new ... end`) On utilise la règle "Séquence", on évalue donc d'abord `t := new array of integer [2]` dans cet état. Pour cela, on utilise "Affectation : variable globale", il nous faut évaluer l'expression `new array of integer [2]`. Pour cela, il faut d'abord évaluer `2`, qui s'évalue en 2 sans changer l'état. On vérifie que 2 est positif, on crée une nouvelle adresse fraîche ℓ dans H, et on arrive donc dans l'état $\{\texttt{t} \mapsto \mathit{nil}\}, \{\ell \mapsto \langle 0, 0\rangle\}, \emptyset$ avec la valeur ℓ. "Affectation : variable globale" mène donc à l'état $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 0, 0\rangle\}, \emptyset$.

Dans cet état, on évalue `t[0] := 1`. On applique la règle "Écriture dans un tableau" : on évalue `t` grâce à "Variable globale" ce qui ne change pas l'état et donne comme valeur ℓ, puis on évalue `0` grâce à "Constante" ce qui ne change pas l'état et produit la valeur 0, puis on évalue `1` grâce à "Constante" ce qui ne change pas l'état et produit la valeur 1. La valeur de ℓ dans le tas de l'état final est $\langle 0, 1\rangle$, et on vérifie que $0 \leq 0 < 2$, donc "Écriture dans un tableau" mène à l'état $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 1, 0\rangle\}, \emptyset$.

Dans cet état, on évalue `swap(t[0],t[1])`. On applique donc la règle "Évaluation des arguments d'une procédure", ce qui nous fait évaluer les arguments de gauche à droite. Pour `t[0]` on applique "Lecture dans un tableau" : `t` a pour valeur ℓ et ne change pas l'état ; `0` s'évalue en 0 et ne change pas l'état ; ℓ est associé à $\langle 1, 0\rangle$ dans le tas ; on vérifie que $0 \leq 0 < 2$; on produit donc la valeur 1 sans changer l'état. De même, `t[1]` produit la valeur 0 sans changer l'état. On doit donc évaluer `swap(1,0)` dans cet état, grâce à "Appel d'une procédure définie". Il faut donc évaluer le corps de `swap` dans l'environnement $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 1, 0\rangle\}, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 0, \texttt{tmp} \mapsto 0\}$.

On applique donc "Séquence", on évalue d'abord `tmp := x` grâce à "Affectation : variable locale" : on évalue `x` via "Variable locale", ce qui produit la valeur 1 sans changer l'état ; et on arrive à l'état $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 1, 0\rangle\}, \{\texttt{x} \mapsto 1, \texttt{y} \mapsto 0, \texttt{tmp} \mapsto 1\}$. Dans cet état, on évalue `x := y`, ce qui mène à l'état $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 1, 0\rangle\}, \{\texttt{x} \mapsto 0, \texttt{y} \mapsto 0, \texttt{tmp} \mapsto 1\}$. Dans cet état on évalue `y := tmp`, ce qui mène à l'état $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 1, 0\rangle\}, \{\texttt{x} \mapsto 0, \texttt{y} \mapsto 1, \texttt{tmp} \mapsto 1\}$. "Appel d'une procédure définie" conduit donc à l'état inchangé $\{\texttt{t} \mapsto \ell\}, \{\ell \mapsto \langle 1, 0\rangle\}, \emptyset$, qui est l'état final du programme.

On note que puisque les appels des fonctions se font par valeur et pas par référence,

les valeurs des cases du tableau ne sont pas échangées.

Comme Pseudo Pascal permet des entrées/sorties (`readln()`), il n'est pas toujours possible de connaître la sémantique d'un programme de façon statique (c'est-à-dire au moment de la compilation). Par exemple, il n'est pas possible de déterminer statiquement si le programme suivant est sémantiquement correct :

```
var t : array of integer
begin
  t :=  new array of integer [2];
  t[readln()] := 1
end
```

Néanmoins, pendant la compilation, il y aura une phase d'analyse sémantique qui restreindra le nombre de programmes qui n'ont pas de sémantique correcte, par exemple par une vérification du typage (qui empêche par exemple l'expression syntaxiquement correcte mais sémantiquement incorrecte `3 + new array of integer [1]`).

6.2. Sémantique de RISC-V

Comme RISC-V est un jeu d'instructions, sa sémantique correspond au fonctionnement des processeurs correspondants. Elle est décrite dans le standard définissant RISC-V. On trouvera en annexe A.3 les instructions les plus courantes. Les processeurs RISC-V possèdent (en général) 32 registres généraux interchangeables, sauf le registre `x0` qui contient toujours la valeur 0. Nous verrons toutefois par la suite que certains registres ont un rôle réservé, notamment pour respecter les conventions d'appel.

Les instructions RISC-V ont globalement les formes suivantes :

- `ins rd, k` où `rd` est le registre de destination et `k` est une constante entière (exemple `li t0, 1`) ;
- `ins rd, a` où `rd` est le registre de destination et `a` est une adresse (exemple `la t0, ma_chaine`) ;
- `ins rd, rs` où `rd` est le registre de destination et `rs` est un registre utilisé (exemple `mv t0, s2`) ;
- `ins rd, rs, k` où `rd` est le registre de destination, `k` est une constante entière et `rs` est un registre utilisé (exemple `addi t0, s2, 3`) ;
- `ins rd, rr, rs` où `rd` est le registre de destination et `rr` et `rs` sont des registres utilisés (exemple `add t0, s2, a1`) ;
- `ins rs, label` où `rs` est un registre utilisé pour une comparaison et `label` la position du code où aller en cas de branchement (exemple `bgtz t0, suite`) ;
- `ins rd, k(rs)` où `ins` est soit `lw` (chargement d'une valeur en mémoire dans un registre) soit `sw` (sauvegarde du contenu d'un registre dans la mémoire), `rd` est le registre à sauvegarder ou à utiliser, le contenu du registre `rs` est l'adresse mémoire à utiliser et `k` est le décalage par rapport à cette adresse ;
- `ecall` (appel système).

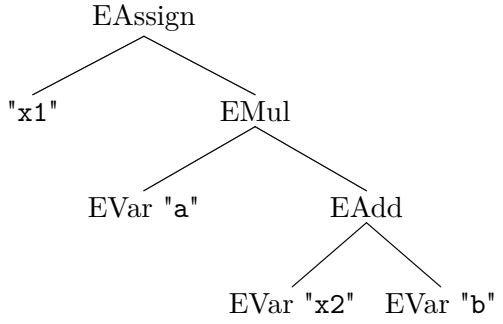
Exemple 6.2 *:* Le code RISC-V suivant calcule la factorielle :

```
f11:
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    mv s0, a0
    blez s0, f4
    addi a0, s0, -1
    jal f11
    mul a0, s0, a0
f16:
    lw ra, 4(sp)
    lw s0, 0(sp)
    addi sp, sp, 8
    ret
f4:
    li a0, 1
    j f16
```

L'évolution des registres lors de l'exécution est laissé en exercice au lecteur. On pourra utiliser un simulateur RISC-V comme par exemple Jupiter https://github.com/andrescv/Jupiter.

7. Analyse syntaxique et analyse sémantique

Le programme passé en entrée du compilateur est une suite de caractères. Avant de commencer à le traduire, il faut commencer par le transformer en une représentation plus structurée qui sera plus facile à manipuler. On passe ainsi de la syntaxe concrète (`x1 := a * (x2 + b)`) à un arbre de syntaxe abstraite



On utilise les méthodes étudiées dans la théorie des langages formels pour arriver à cette fin. En général, le but est d'arriver à reconnaître un langage formel défini à l'aide d'une grammaire hors contexte. Néanmoins, en pratique, on a parfois besoin de connaître le contexte, par exemple pour connaître le type d'une variable définie plus tôt.

Il est rare maintenant d'écrire un analyseur syntaxique à la main. En général, on utilise des outils qui permettent de générer automatiquement des analyseurs à partir d'une spécification du langage.

Généralement, on procède en deux phases : la première découpe l'entrée en séquence de mots élémentaires, les lexèmes ou *token*, en cherchant à reconnaître un langage régulier, alors que la seconde reconnaît un langage hors contexte sur l'alphabet composé de ces lexèmes. On parle d'analyse lexicale pour la première et d'analyse syntaxique pour la seconde.

Il est évident qu'un langage régulier n'est pas suffisant pour obtenir un langage de programmation de haut niveau satisfaisant. (Rappelons que le langage des expressions bien parenthésées n'est pas régulier.) On pourrait se contenter de la deuxième passe, puisque tout langage régulier est hors contexte. On parle alors de *scannerless parsing*. Néanmoins, les techniques de génération d'analyseurs à partir d'une spécification (c'est-à-dire à partir d'une expression régulière ou d'une grammaire) fournissent des analyseurs plus efficaces dans le cas des langages réguliers, ce qui justifie leur emploi. La construction de la grammaire hors contexte en est par ailleurs simplifiée.

Bien que conceptuellement séparées, analyses lexicale et syntaxique sont habituellement « pipelinées ». L'analyseur lexical fournit chaque lexème sur demande de l'analyseur syntaxique, ce qui évite de construire en mémoire l'intégralité de la suite de lexèmes. Les deux analyses sont donc exécutées de façon entremêlée.

7.1. Analyse lexicale

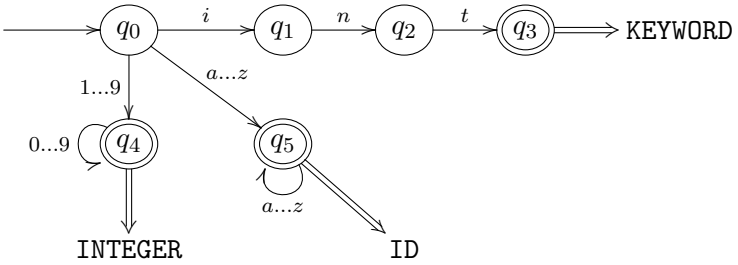
Le but est de reconnaître une séquence de mots appartenant à un langage défini à l'aide d'une expression régulière. On utilise pour cela des techniques utilisant des automates finis.

On part d'une expression régulière de la forme $e_1|\dots|e_n$ où à chaque e_i est associé un lexème à produire. On transforme l'expression régulière en automate fini non déterministe avec ϵ-transitions, chaque état final de l'automate correspondant à la reconnaissance d'un lexème. Ensuite, on déterminise cet automate, on en calcule l'ϵ-fermeture et on le minimise. Pour implémenter l'automate ainsi obtenu, on utilise souvent un tableau qui contient la fonction de transition.

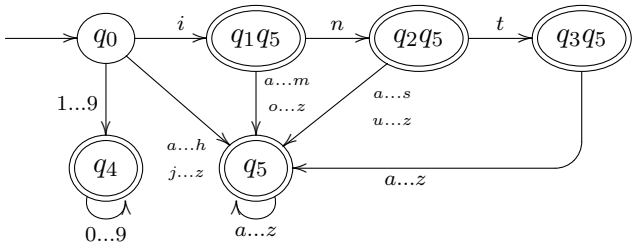
Exemple 7.1 *:* On considère l'expression régulière avec productions suivante :

```
int             return(KEYWORD);
[a-z]+          return(ID);
[1-9][0-9]*     return(INTEGER);
```

Ceci correspond à l'automate fini non déterministe



Une fois déterminisé, celui-ci devient



Toutefois, l'analyse lexicale ne se contente pas de reconnaître les mots appartenant au langage défini par une certaine expression régulière. Elle produit également une suite de lexèmes, un pour chaque mot reconnu, qui sera ensuite utilisée par l'analyseur syntaxique.

Une fois l'automate déterminisé, la reconnaissance de la séquence de lexème peut être ambiguë. Par exemple, si le langage à reconnaître est $a+$, alors partant de aa on peut soit reconnaître un lexème aa, soit reconnaître une séquence de deux lexèmes a et a. Dans les générateurs d'analyseurs lexicaux comme lex, cela est résolu de la façon suivante : on cherche par défaut à reconnaître le plus long préfixe de l'entrée possible, et si deux expressions régulières reconnaissent le même mot, c'est le lexème correspondant à celle écrite en premier qui est choisi. En pratique, cela revient à dire que l'automate doit continuer tant que cela est possible, et revenir au dernier état final atteint en cas d'erreur, et que dans l'automate déterminisé, le lexème reconnu par un état final est celui correspondant à l'expression reconnue par cet état définie en premier.

Exemple 7.2 *:* Dans l'automate déterminisé de l'exemple 7.1, les états finaux q_1q_5, q_2q_5 et q_5 produisent `ID`, tandis que q_3q_5 produit `KEYWORD`, l'expression `int` étant définie avant `[a-z]+` dans la spécification. Sur l'entrée *integer*, l'analyseur produira le lexème `ID` tandis que sur *int* il produira `KEYWORD`. Sur *int*38*er*, l'analyseur produira la séquence de lexèmes `KEYWORD INTEGER ID`.

Comme on reconnaît les plus longs préfixes, il faut faire attention aux expressions `.*` ; par exemple, si on cherche à reconnaître les mots entre guillemets simples, il ne faut pas utiliser l'expression `'.*'` car `'a' + 'b'` sera reconnu comme un seul lexème : la chaîne `a' + 'b` contenue entre guillemets simples. À la place, on peut utiliser `'[^']*'` (cf. la signification des expressions régulières de lex en annexe A.1).

7.2. Analyse syntaxique

On rappelle les définitions de bases des grammaires hors contexte.

Définition 7.1 (Grammaire hors contexte)**.** *Une grammaire hors contexte (ou grammaire algébrique, ou grammaire non contextuelle) est donnée par un quadruplet* (Σ, V, S, P) *où :*

- Σ *est l'alphabet des symboles terminaux, notés a, b, etc. Les symboles terminaux sont typiquement les lexèmes produits par l'analyse lexicale ;*
- V *est un ensemble de symbole non terminaux, notés A, B, etc., disjoint de* Σ *;*
- $S \in V$ *est le symbole de départ ;*
- P *est un ensemble de production de la forme* $A \to w$*, où* w *est un mot sur* $\Sigma \cup V$*.*

Exemple 7.3 *:* On peut utiliser la grammaire G_A suivante pour définir des expressions arithmétique :

- $\Sigma = \{int, (,), +, -, \times, /\}$;
- $V = \{E\}$;
- $S = E$;

— $P = \begin{Bmatrix} E \to E + E \\ E \to E - E \\ E \to E \times E \\ E \to E/E \\ E \to (E) \\ E \to int \end{Bmatrix}$

Définition 7.2 (Langage défini par une grammaire)**.** *Étant donné des mots u et v sur $\Sigma \cup V$ et une production $A \to w$, on dit que uwv peut être dérivé à partir de uAv, et on note $uAv \longrightarrow uwv$. $\longrightarrow^+$ désigne la fermeture transitive de $\longrightarrow$.*

Une grammaire G défini un langage $\mathcal{L}(G)$ sur l'alphabet Σ dont les éléments sont les mots engendrés par dérivation à partir du symbole de départ S :

$$\mathcal{L}(G) := \{w \in \Sigma^* \mid S \longrightarrow^+ w\}$$

Exemple 7.4 *:* $int + int * int \in \mathcal{L}(G_A)$ comme le montre les dérivations suivantes :

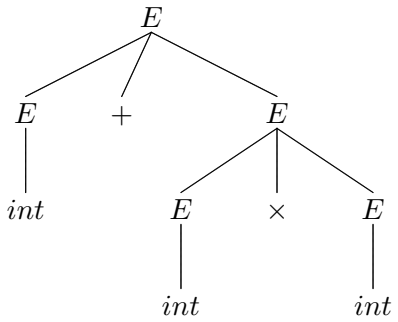
$E \longrightarrow E + E \longrightarrow E + E * E \longrightarrow int + E * E \longrightarrow int + int * E \longrightarrow int + int * int$
$E \longrightarrow E + E \longrightarrow int + E \longrightarrow int + E * E \longrightarrow int + int * E \longrightarrow int + int * int$
$E \longrightarrow E * E \longrightarrow E + E * E \longrightarrow int + E * E \longrightarrow int + int * E \longrightarrow int + int * int$

Les deux premières dérivations correspondent à la même façon de comprendre le mot, en voyant + reconnu plus haut que ×, ce qui n'est pas le cas pour la dernière.

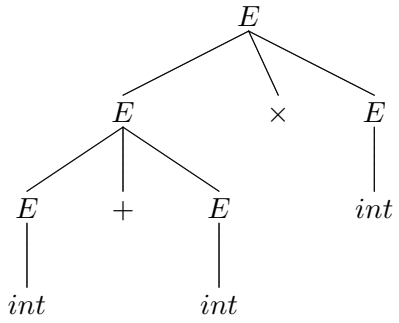
Pour mieux distinguer les dérivations, on les représente sous forme d'arbre dont la racine est S et où les fils d'un nœud non terminal sont les symboles de la partie droite de la production utilisée dans la dérivation.

Certaines dérivations possèdent alors le même arbre : les productions y ont été appliquées en parallèle dans des ordres différents.

Exemple 7.5 *:* L'arbre correspondant aux deux premières dérivations est le suivant :



Celui correspondant à la dernière est :



Définition 7.3 (Ambiguïté)**.** *Une grammaire est dite ambiguë s'il existe un mot de son langage reconnu par deux dérivations avec des arbres différents.*

Pour le compilateur (et surtout son utilisateur) il est important d'avoir des grammaires non ambiguës, car les arbres de dérivations vont être utile pour obtenir la syntaxe abstraite du programme. Étant donnée une expression du langage, on ne veut avoir qu'une façon de la comprendre. Par conséquent, on ne considère que des grammaires non ambiguës. Comme il est indécidable de savoir si une grammaire est ambiguë, il faut même se restreindre à des sous-classes de l'ensemble des grammaires non ambiguës.

Toutefois, l'ambiguïté est propriété de la grammaire, et non pas du langage. Par conséquent, il est souvent possible de modifier la grammaire pour la rendre non ambiguë tout en reconnaissant le même langage, en utilisant par exemple des priorités pour les opérateurs et de l'associativité.

Exemple 7.6 *:* La grammaire suivant G'_A reconnaît le même langage que G_A, mais n'est pas ambiguë :

— $\Sigma = \{int, (,), +, -, \times, /\}$;
— $V = \{E, T, F\}$;
— $S = E$;
— $P = \begin{Bmatrix} E \to E + T \\ E \to E - T \\ E \to T \\ T \to T \times F \\ T \to T/F \\ T \to F \\ F \to (E) \\ F \to int \end{Bmatrix}$

7.2.1. Analyse decendante, analyse LL(1)

L'analyse LL(1) est une analyse de type récursive descendante, ce qui est le plus simple conceptuellement. Il s'agit de développer le non-terminal le plus à gauche jusqu'à tomber sur un terminal. On compare alors ce terminal avec le premier symbole du flux de lexème. Si c'est le même, on passe à la suite du mot, sinon il faut revenir en arrière (*backtracker*)

en utilisant d'autres productions. Pour limiter ces retours en arrière, dans LL(1) on est autorisé à regarder un symbole du flux de lexème au moment de choisir quelle règle de production choisir. LL(1) signifie en fait *Left-to-right scanning, Leftmost derivation, use 1 symbol lookahead*, c'est-à-dire que le flux de lexème est parcouru de gauche à droite, que l'on cherche la dérivation la plus à gauche, en ayant accès à un symbol du flux de lexème.

Exemple 7.7 *:* Avec la grammaire G'_A, on peut essayer de reconnaître $int + int \times int$ de la façon suivante :

$\underline{E}$	$\bullet int + int \times int$
$\underline{E} + T$	$\bullet int + int \times int$
$\underline{T} + T$	$\bullet int + int \times int$
$\underline{F} + T$	$\bullet int + int \times int$
$\underline{int} + T$	$\bullet int + int \times int$
$int \underline{+} T$	$int \bullet + int \times int$
$int + \underline{T}$	$int + \bullet int \times int$
$int + \underline{T} \times F$	$int + \bullet int \times int$
$int + \underline{F} \times F$	$int + \bullet int \times int$
$int + \underline{int} \times F$	$int + \bullet int \times int$
$int + int \underline{\times} F$	$int + int \bullet \times int$
$int + int \times \underline{F}$	$int + int \times \bullet int$
$int + int \times \underline{int}$	$int + int \times \bullet int$
$int + int \times int$	$int + int \times int \bullet$

où $\bullet$ désigne la position courante dans le flux de lexèmes. Ici, on a fait les bons choix de règle à chaque fois. Si on avait commencé par

$\underline{E}$	$\bullet int + int \times int$
$\underline{T}$	$\bullet int + int \times int$
$\underline{T} \times F$	$\bullet int + int \times int$
$\underline{F} \times F$	$\bullet int + int \times int$
$\underline{int} \times F$	$\bullet int + int \times int$
$int \underline{\times} F$	$int \bullet + int \times int$

alors on aurait été obligé de backtracker.

D'autre part, on peut ne jamais terminer en faisant par exemple

$\underline{E}$	$\bullet int + int \times int$
$\underline{E} + T$	$\bullet int + int \times int$
$\underline{E} + T + T$	$\bullet int + int \times int$
$\underline{E} + T + T + T$	$\bullet int + int \times int$
$\vdots$	
$\underline{E} + T + T + \ldots + T$	$\bullet int + int \times int$
$\vdots$	

Comme on le voit, deux problèmes principaux se posent. Premièrement, dans le cas où on a des règles récursives à gauche (par exemple $A \to Aa$), on peut ne jamais arriver jusqu'à un terminal. Deuxièmement, on veut limiter au maximum d'avoir à backtracker en regardant le premier lexème pour choisir les règles à appliquer.

Élimination de la récursivité à gauche

Une grammaire est récursive à gauche s'il existe un non terminal A et une dérivation $A \longrightarrow^+ Aw$.

Il est possible de transformer mécaniquement une grammaire pour enlever toute récursivité à gauche, en autorisant des productions dont le membre droit est le mot vide ϵ.

Si on a une grammaire dont les productions avec comme membre gauche A sont $A \to Aa$ et $A \to b$, alors les mots reconnus par A sont b suivis d'une liste éventuellement vide de a. Par conséquent, on peut remplacer ces productions par $A \to bA'$, $A' \to aA'$ et $A' \to \epsilon$. On peut généraliser cette idée à toutes les grammaires récursives à gauche. Premièrement, si la récursivité est indirecte, on déroule les règles pour obtenir une récursivité directe. Par exemple, si on part de

$$\begin{array}{l} A \to Ba \\ B \to Ab \\ B \to c \end{array} \qquad \text{on la transforme en} \qquad \begin{array}{l} A \to Aba \\ A \to ca \\ B \to Ab \\ B \to c \end{array} .$$

Ensuite, les règles de production pour A sont soit de la forme $A \to Aw$ soit $A \to v$ où on ne peut pas dériver un mot de la forme Au à partir de v. On transforme

$$\begin{array}{l} A \to Aw_1 \\ \quad \vdots \\ A \to Aw_n \\ A \to v_1 \\ \quad \vdots \\ A \to v_m \end{array} \qquad \text{en} \qquad \begin{array}{l} A \to v_1 A' \\ \quad \vdots \\ A \to v_m A' \\ A' \to w_1 A' \\ \quad \vdots \\ A' \to w_n A' \\ A' \to \epsilon \end{array} .$$

Exemple 7.8 *:* Si on applique ceci sur G'_A, on obtient la grammaire G''_A dont les produc-

tions sont

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \\
E' &\to -TE' \\
E' &\to \epsilon \\
T &\to FT' \\
T' &\to \times FT' \\
T' &\to /FT' \\
T' &\to \epsilon \\
F &\to (E) \\
F &\to int
\end{aligned}
$$

Factorisation gauche

Dans LL(1), on veut pouvoir choisir quelle production utiliser en regardant uniquement le prochain lexème disponible. Si on a par exemple la grammaire avec les productions suivantes

$$
\begin{aligned}
I &\to if\ C\ then\ I\ else\ I\ fi \\
I &\to if\ C\ then\ I\ fi \\
I &\to a
\end{aligned}
$$

il n'est pas possible en regardant le premier lexème de savoir s'il faut choisir la première ou la deuxième règle de production si c'est un if. La solution est de factoriser le préfixe commun pour n'avoir qu'une seule production pour les deux cas. Sur l'exemple, cela donne

$$
\begin{aligned}
I &\to if\ C\ then\ I\ FIN\_IF \\
I &\to a \\
FIN\_IF &\to else\ I\ fi \\
FIN\_IF &\to fi
\end{aligned}
$$

Plus généralement, pour chaque non-terminal A, on recherche le plus grand préfixe commun w à au moins deux des membres droits des productions de A. Les production de A

s'écrivent alors

$$\begin{aligned} A &\to wv_1 \\ &\vdots \\ A &\to wv_n \\ A &\to u_1 \\ &\vdots \\ A &\to u_m \end{aligned}$$

où w n'est préfixe d'aucun $u_1, \ldots, u_m$. On transforme ces productions en

$$\begin{aligned} A &\to wA' \\ A &\to u_1 \\ &\vdots \\ A &\to u_m \\ A' &\to v_1 \\ &\vdots \\ A' &\to v_n \end{aligned}$$

où A' est un non-terminal frais, et on recommence jusqu'à ce qu'il n'y ait plus de préfixe commun à au moins deux membres droits.

Une fois ces transformations effectuées, s'il est possible de savoir quelle règle de production appliquer à tout instant sans avoir besoin de backtracker et sans regarder le flux de lexèmes, on dit que la grammaire est LL(0). Malheureusement, les grammaires LL(0) sont sans utilité pratique : puisqu'il n'y a pas besoin du flux de lexème pour trouver la dérivation, un seul mot peut être dérivé à partir du symbole de départ. Le langage d'une grammaire LL(0) est donc un singleton. Pour avoir des langages plus riche, on veut utiliser les symboles du flux de lexème pour décider quelle règle appliquer. Dans LL(1), on regardera le premier lexème uniquement.

Ensemble First et Follow

Un analyseur LL(1) doit pouvoir décider, étant donnés un non-terminal A à développer et un premier lexème lu a, quelle production appliquer.

Il y a deux cas : soit il existe une production $A \to w$ avec $w \longrightarrow^* av$, soit il existe une production $A \to w$ avec $w \longrightarrow^* \epsilon$ et le terminal a peut suivre un A dans les mots reconnus par la grammaire. Il nous faut donc calculer les deux ensembles suivants :

Définition 7.4 (Prédicat *Nullable*, ensembles *First*, *Follow*)**.** *Étant donné un mot $w \in (\Sigma \cup V)^*$, le prédicat $Nullable(w)$ est défini comme vrai si le mot vide peut être*

dérivé de w :

$$Nullable(w) = \begin{cases} vrai & si\ w \longrightarrow^* \epsilon \\ faux & sinon \end{cases}$$

Étant donné un mot $w \in (\Sigma \cup V)^*$, l'ensemble $First(w)$ est l'ensemble des terminaux qui peuvent apparaître comme premier symbole dans une phrase dérivée du mot w :

$$First(w) = \{a \in \Sigma \mid w \longrightarrow^* av\}$$

Étant donné un non-terminal A, l'ensemble $Follow(A)$ est l'ensemble des symboles terminaux qui peuvent apparaître immédiatement après un A dans une phrase valide du langage :

$$Follow(A) = \{a \in \Sigma \mid S \longrightarrow^+ uAav\}$$

Étant donné une grammaire, on peut calculer ces ensembles en temps polynomial à l'aide d'un algorithme itératif. En effet, $First$ et $Follow$ sont les solutions des inéquations suivantes :

$Nullable(a) \sqsupseteq faux$	
$Nullable(A) \sqsupseteq Nullable(w)$	pour tout w tel que $A \to w \in P$
$Nullable(\epsilon) \sqsupseteq vrai$	
$Nullable(sw) \sqsupseteq Nullable(s) \wedge Nullable(w)$	
$First(a) \supseteq \{a\}$	
$First(A) \supseteq First(w)$	pour tout w tel que $A \to w \in P$
$First(\epsilon) \supseteq \emptyset$	
$First(sw) \supseteq First(s)$	
$First(sw) \supseteq First(w)$	si $Nullable(s)$
$Follow(B) \supseteq First(v)$	si $A \to uBv \in P$
$Follow(B) \supseteq Follow(A)$	si $A \to uBv \in P$ et $Nullable(v)$

Pour que l'analyse LL(1) fonctionne, pour tout non-terminal A, si les productions partant de A sont

$$\begin{aligned} A &\to w_1 \\ &\vdots \\ A &\to w_n \\ A &\to \epsilon \end{aligned}$$

il faut que les ensembles $First(w_1)$, ..., $First(w_n)$ et $Follow(A)$ soient disjoints deux-à-deux (sans regarder $Follow(A)$ si on n'a pas de production vers ϵ). Si le non-terminal

à réduire est A et que le premier symbole du flux de lexème est a, on appliquera donc la règle $Arightarrow w_i$ si a est dans $First(w_i)$ ou la règle $A \to \epsilon$ si a est dans $Follow(A)$. Dans le cas contraire, on aura une erreur de syntaxe (le flux de lexème n'appartient pas au langage de la grammaire) que l'on pourra reporter.

Quand les ensembles $First$ et $Follow$ sont disjoints comme décrit ci-dessus, on dit que la grammaire appartient à la classe LL(1). LL(1) est une sous-classe stricte des grammaires non ambiguës. Cette sous-classe est décidable (il suffit de calculer les ensembles $First$ et $Follow$ et de vérifier qu'ils sont disjoints), mais étant donné un langage $\mathcal{L}(G)$ défini par une grammaire hors contexte G, il est indécidable de savoir si $\mathcal{L}(G)$ peut être défini par une grammaire LL(1) (éventuellement différente de G).

Exemple 7.9 *:* Pour la grammaire G''_A, on a les ensembles $First$ et $Follow$ suivants :

	$Nullable$
T	$faux$
T'	$vrai$
F	$faux$
F'	$vrai$

	$First$
TE'	int (
$+TE'$	$+$
$-TE'$	$-$
FT'	int (
$\times FT'$	$\times$
$/FT'$	$/$
(E)	(
int	int
E'	$+ -$
T'	$\times /$

	$Follow$
E	)
E'	)
T	$+ -)$
T'	$+ -)$
F	$\times / + -)$

On peut vérifier que les ensembles $First$ des membres droits des productions partant du même non-terminal sont disjoints deux-à-deux, et également disjoints de l'ensemble $Follow$ de ce non-terminal dans le cas où il y a une production vers le mot vide. G''_A est donc une grammaire LL(1).

Exercice 7.1 *:* Retrouvez ces valeurs en utilisant l'algorithme itératif.

Une fois les ensembles $First$ et $Follow$ calculés, il est facile de construire les fonctions analysant les non-terminaux.

Exemple 7.10 *:* Pour la grammaire G''_A on obtient les fonctions suivantes :

```
fonction decale()
  mot_courant <- lexeme_suivant();
  vrai

fonction E()          fonction E'()
  T() && E'()           filtre mot_courant avec
                            '+' -> decale() && T() && E' ()
                          | '-' -> decale() && T() && E' ()
```

```
                          | _ -> vrai   (* E' → ϵ *)

fonction T()          fonction T'()
  F() && T'()           filtre mot_courant avec
                            '×' -> decale() &&  F() && T' ()
                          | '/' -> decale() && F() && T' ()
                          | _ -> vrai  (* T' → ϵ *)

fonction F()
  filtre mot_courant avec
      '(' -> decale() && E() && mot_courant = ')' && decale()
    | 'int' -> decale()
    | _ -> faux

fonction main()
  decale(); E() && mot_courant = 'EOF'
```

Il est également possible d'imaginer une implémentation à l'aide de tableaux indiquant, pour chaque non-terminal, la règle à appliquer en fonction du lexème rencontré.

En résumé, l'analyse LL(1)

— est conceptuellement simple : on peut presque transcrire manuellement une grammaire LL(1) en un analyseur ;
— produit des analyseurs compacts et efficaces, dans lesquels il est facile d'ajouter des fonctionnalités de débogages pour les entrées non acceptées ;
— mais nécessite des transformations de la grammaire initiale si elle présente des facteurs à gauche ou de la récursivité à gauche ;
— certains langages assez naturels ne possèdent pas de grammaire LL(1).

Cette approche est toutefois utilisée dans certains générateurs d'analyseurs comme JavaCC [1] ou ANTLR [2] (qui utilisent en fait des extensions où il est par exemple possible de vérifier si les prochains lexèmes appartiennent à un certain langage régulier).

7.2.2. Analyse ascendante, analyse LR

L'approche LR est basée sur les automates à piles. Il s'agit d'empiler les lexèmes comme feuilles de l'arbre de dérivation, jusqu'à être capable d'appliquer une production $A \to w$, auquel on dépile $|w|$ éléments et on empile A. Comme on remonte dans l'arbre de dérivation, on construit la dérivation à l'envers, d'où le nom *Left-to-right scanning, Reverse-rightmost derivation*, avec ici aussi la possibilité de regarder n symboles du flux de lexèmes pour LR(n). On construit d'abord un automate non déterministe qu'il s'agit ensuite de déterminiser.

1. https://javacc.dev.java.net/
2. http://www.antlr.org/

LR(0)

Pour construire l'automate permettant de reconnaître une grammaire LR(0), on va considérer deux types d'états possible : $\bullet A$ signifiera « je m'apprête à reconnaître A », et $A \to u \bullet v$ signifiera « j'ai déjà reconnu u, j'ai encore besoin de reconnaître v pour avoir une réduction de A ».

On a trois sortes de transitions :

— Pour chaque non-terminal A dont les productions sont

$$\begin{aligned} A &\to w_1 \\ &\vdots \\ A &\to w_n \end{aligned}$$

on ajoute des ϵ-transitions entre $\bullet A$ et les $A \to \bullet w_i$.
— Étant donné un état $A \to u \bullet sw$ avec $s \in \Sigma \cup V$ on ajoute une s-transition vers $A \to us \bullet w$.
— On ajoute des ϵ-transitions entre un état $A \to u \bullet Bv$ et $\bullet B$.

L'automate maintient une pile d'états, dont le sommet constitue l'état courant, et a accès à un flux de lexèmes. Il y a deux actions possibles :

décaler (*shift*) : si le lexème de tête est a, alors on peut retirer a du flux et empiler un nouvel état s' accessible à partir de s à travers un chemin étiqueté par $\epsilon^* a$;

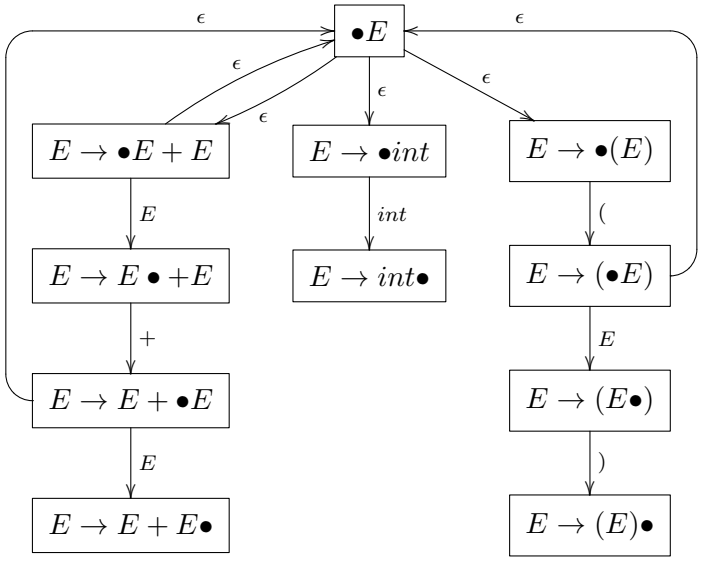
réduire (*reduce*) : si l'état courant est étiqueté $A \to w\bullet$, on peut dépiler $|w|$ éléments pour arriver à un état courant s_0 et ajouter A en tête du flux d'entrée.

Le mot est accepté si la pile contient uniquement l'état $\bullet E$ et le flux d'entrée contient uniquement E.

Exemple 7.11 *:* Pour la grammaire G_+ :

$$E \to E + E \tag{7.1}$$
$$E \to (E) \tag{7.2}$$
$$E \to int \tag{7.3}$$

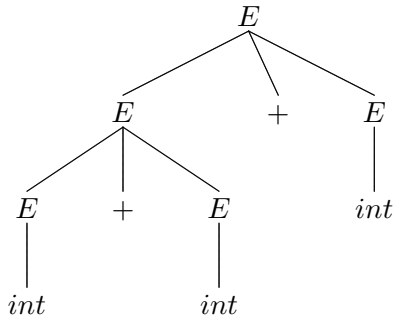
l'automate non déterministe est le suivant :



En partant de $int + int + int$ on a les deux exécutions suivantes possibles :

	action	pile	flux
		$\bullet E$	$int + int + int$
1	décaler int	$\bullet E \vert E \to int\bullet$	$+int + int$
2	réduire (7.3)	$\bullet E$	$E + int + int$
3	décaler E	$\bullet E \vert E \to E \bullet + E$	$+int + int$
4	décaler $+$	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	$int + int$
5	décaler int	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to int\bullet$	$+int$
6	réduire (7.3)	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	$E + int$
7	décaler E	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E + E\bullet$	$+int$
8	réduire (7.1)	$\bullet E$	$E + int$
9	décaler E	$\bullet E \vert E \to E \bullet + E$	$+int$
10	décaler $+$	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	int
11	décaler int	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to int\bullet$	
12	réduire (7.3)	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	E
13	décaler E	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E + E\bullet$	
14	réduire (7.1)	$\bullet E$	E
	accepté		

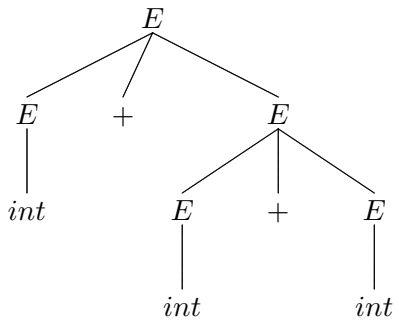
qui produit l'arbre de dérivation



et d'autre part

	action	pile	flux
		$\bullet E$	$int + int + int$
1	décaler int	$\bullet E \vert E \to int\bullet$	$+int + int$
2	réduire (7.3)	$\bullet E$	$E + int + int$
3	décaler E	$\bullet E \vert E \to E \bullet + E$	$+int + int$
4	décaler $+$	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	$int + int$
5	décaler int	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to int\bullet$	$+int$
6	réduire (7.3)	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	$E + int$
$7'$	décaler E	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E \bullet + E$	$+int$
$8'$	décaler $+$	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	int
$9'$	décaler int	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to int\bullet$	
$10'$	réduire (7.3)	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	E
$11'$	décaler$E$	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E + E\bullet$	
$12'$	réduire (7.1)	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E$	E
$13'$	décaler E	$\bullet E \vert E \to E \bullet + E \vert E \to E + \bullet E \vert E \to E + E\bullet$	
$14'$	réduire (7.1)	$\bullet E$	E
	accepté		

correspondant à la dérivation qui produit l'arbre de dérivation



On voit que les dérivations divergent à partir de l'étape 8, où il est possible de faire soit une réduction soit un décalage. On parle alors de conflit décaler/réduire. Il existe également des conflits réduire/réduire, quand deux règles de réductions pourraient s'ap-

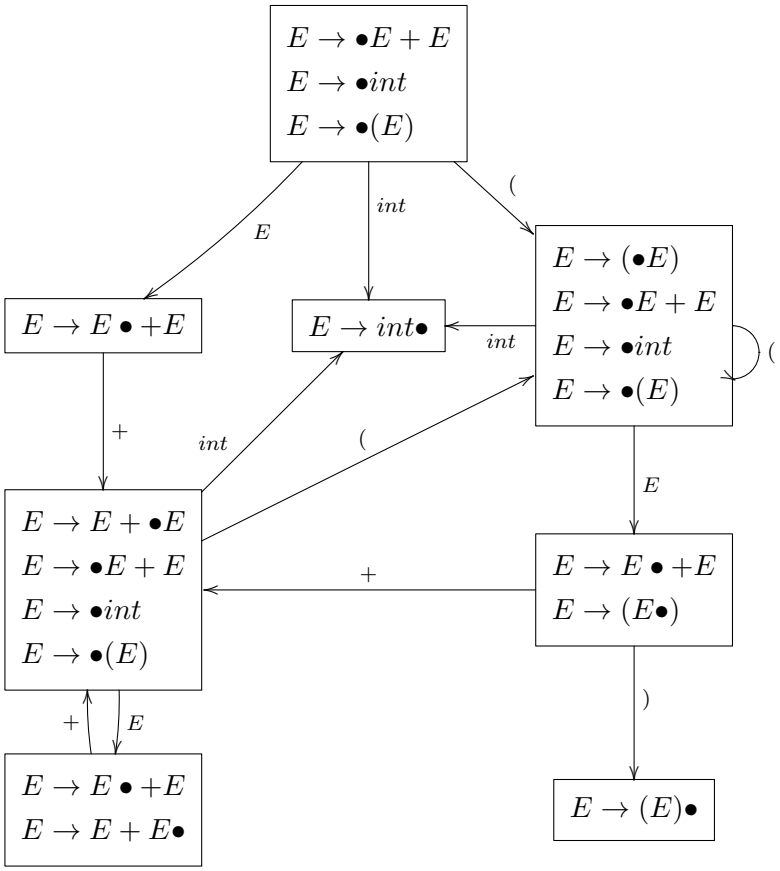
pliquer. Un exemple de grammaire avec un conflit réduire/réduire est G_{ab} :

$$E \to Ab \tag{7.4}$$
$$E \to A'b \tag{7.5}$$
$$A \to a \tag{7.6}$$
$$A' \to a \tag{7.7}$$

Sur l'entrée ab, après avoir décalé a, on ne sait pas si on doit réduire par (7.6) ou par (7.7).

Les conflits de la grammaire apparaissent lors de la déterminisation de l'automate. Pour cela, on utilise la technique classique de *powerset construction*.

Exemple 7.12 *:* En déterminisant l'automate pour la grammaire G_+, on obtient l'automate suivant (on n'indique pas les états $\bullet E$ car ils sont forcément suivis uniquement d'ϵ-transitions)



On voit un conflit dans l'état $\boxed{\begin{array}{l} E \to E \bullet + E \\ E \to E + E \bullet \end{array}}$ où il est possible de décaler avec + ou de réduire avec $E \to E + E$.

Exercice 7.2 *:* Construire l'automate non déterministe pour G_{ab}, le déterminiser. Où se trouve(nt) le(s) conflit(s) ?

LR(1)

Considérons la grammaire G_{list} suivante :

$$L \to aL \tag{7.8}$$
$$L \to a \tag{7.9}$$

Bien que non ambiguë, elle possède un conflit décaler/réduire dans l'état $\boxed{\begin{array}{l} L \to \bullet\, aL \\ L \to \bullet\, a \\ L \to a \bullet L \\ L \to a \bullet \end{array}}$.

Pour savoir s'il faut réduire par (7.9) ou décaler a, il faut savoir si on a atteint la fin de la liste ou pas, et donc prendre en compte un symbole de prévision. C'est la technique LR(1), dans laquelle les états sont de la forme $A \to u \bullet v[a]$ ce qui signifie « j'ai déjà reconnu u, j'ai encore besoin de reconnaître v pour avoir une réduction de A à condition que le lexème suivant soit alors a ». On rajoute toujours un terminal spécial \$ signifiant qu'on a atteint la fin du flux de lexèmes.

On dira qu'il y a un conflit dans l'automate déterminisé quand deux actions y sont possibles dans le même état *pour le même lexème*.

Dans le cas de G_{list} on obtient le même automate accessible à partir de l'état initial, mais avec des états $L \to u \bullet v[\$]$ au lieu de $L \to u \bullet v$. Il n'y a toutefois plus de conflit décaler/réduire, car le décalage ne se fera que si le flux de lexème commence par a alors que la réduction ne se fera que si le flux est vide.

On dira qu'une grammaire appartient à la classe LR(1) ssi l'automate obtenu de cette façon ne présente aucun conflit. La classe LR(1) contient strictement la classe LL(1). Toutefois, la construction LR(1) est coûteuse en taille en pratique. En effet, le nombre d'états de l'automate peut être beaucoup plus important que pour LR(0). Par conséquent, de nombreux générateurs d'analyseurs utilisent des approximations : SLR(1) et LALR(1). Dans ces approximations, on construit le même automate que pour LR(0), mais on calcule également pour les réductions des ensembles $Follow$ ou $Look - Ahead$ qui permettent de savoir si on doit réduire ou pas. La classe des grammaires reconnues par ces approximations est strictement incluse dans celle de LR(1). Par conséquent, des conflits qui peuvent sembler artificiels peuvent apparaître.

Les générateurs d'analyseurs les plus connus sont basés sur LALR(1) : yacc, bison, ocamlyacc, etc. François Pottier et Yann Régis-Gianas ont développé un générateur d'analyseur basé sur une version de LR(1) produisant des analyseurs plus efficaces, menhir [3].

3. http://pauillac.inria.fr/~fpottier/menhir/

Résolution des conflits

Une grammaire ambiguë ne peut évidemment pas être dans LR(1). Nous avons vu avec l'exemple de G_+ que cela se traduisait sous forme de conflit, chacune des actions de ce conflit conduisant à une dérivation différente. Pour résoudre ces conflits, il est bien entendu possible de transformer la grammaire pour la rendre non ambiguë. Une autre solution est d'indiquer manuellement si l'automate doit préférer réduire ou décaler.

Cette seconde solution sort du cadre formel des grammaires hors contexte, mais offre plus de confort. Dans le cas de G_+, on a vu que si on préfère réduire, l'opérateur $+$ devient associatif à gauche, tandis que si on préfère décaler, l'opérateur devient associatif à droite. Dans les générateurs d'analyseur à la yacc, on a donc des mots clefs `%left` et `%right` qui permettent d'indiquer qu'en cas de conflit la dérivation gauche ou droite doit être préférée. En pratique, cela fonctionne de la manière suivante : à certains terminaux est associée une précédence, avec éventuellement une indication d'associativité à gauche ou à droite. Deux terminaux avec la même précédence ont la même associativité (ou absence d'associativité). La précédence et l'associativité d'une production est alors celle de son dernier litéral (mais elle peut être surchargée à l'aide du mot clef `%prec`). Lors d'un conflit décaler/réduire, on essaie de comparer la précédence de la production à utiliser pour réduire avec celle du lexème à décaler. Si l'une d'elles n'est pas définie, le conflit est reporté. Si l'une des précédences est plus grande que l'autre, on exécute l'action en faveur de celle-ci, c'est-à-dire qu'on réduit si la précédence de la production est la plus grande, et on décale si c'est celle du terminal. Si les précédences sont égales, on utilise l'associativité, en réduisant si l'associativité est à gauche, en décalant si elle est à droite, et en reportant le conflit si elle n'est pas définie.

Exemple 7.13 *:* Pour rendre la grammaire G_A non ambiguë et obtenir le même résultat qu'avec la grammaire G'_A, on peut utiliser les précédences suivantes :

Terminal	Précédence	Associativité
$+$ $-$	1	gauche
$\times$ /	2	gauche
( ) int	non définies	

Récursivité

Contrairement à LL(1), LR(1) est capable de reconnaître des grammaires récursives à gauche comme à droite. Toutefois, comme on empile les états tant qu'aucune réduction n'est faite, il est plus efficace de reconnaître les grammaires récursives à gauche, comme l'illustre l'exemple suivant.

Exemple 7.14 *:* Le langage des séquences de a peut être définit par la grammaire récursive à gauche G_g

$$L \to La \tag{7.10}$$
$$L \to a \tag{7.11}$$

comme par la grammaire récursive à droite G_d

$$L \to aL \tag{7.12}$$
$$L \to a \tag{7.13}$$

qui sont toutes deux dans LR(1).

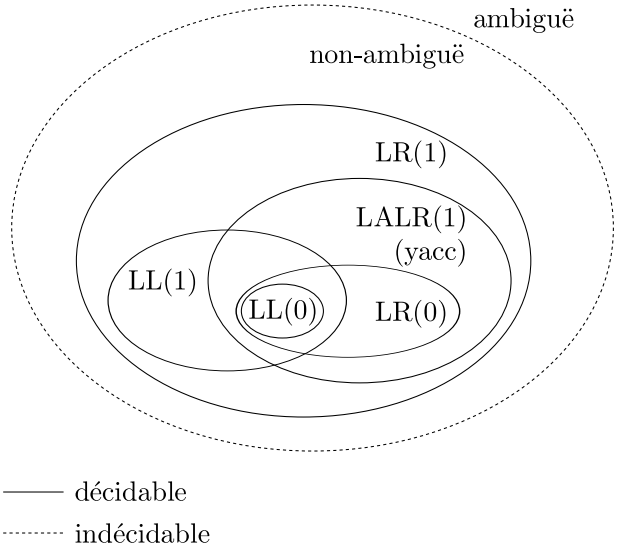
Pour reconnaître $aa \ldots a$, l'automate pour G_g va d'abord décaler a, réduire avec (7.11), décaler a, réduire avec (7.10), ..., décaler a, réduire avec (7.10). La pile contiendra donc au maximum deux états. L'automate pour G_d va quant à lui décaler a, décaler a, ..., décaler a, réduire avec (7.12), ..., réduire avec (7.12), réduire avec (7.13). Par conséquent, la pile de l'automate pour G_d va grossir autant que l'entrée. Il est donc préférable, dans la mesure du possible, d'utiliser la récursivité à gauche dans un parser LR.

7.2.3. Comparaison des analyses

Les techniques LL(n) et LR(n) définissent des sous-classes de la classe des grammaires non ambiguës, et par la même des sous-classes des langages hors-contexte. Une grammaire LL(1) est également LR(1), mais pas réciproquement (grammaire récursive à gauche par exemple). Une grammaire LL(1) n'est par contre par forcément LALR(1) (c'est-à-dire reconnue par yacc), comme le montre l'exemple suivant :

$$\begin{aligned}
S &\to aA \\
&\mid bB \\
A &\to Ca \\
&\mid Db \\
B &\to Cb \\
&\mid Da \\
C &\to E \\
D &\to E \\
E &\to \epsilon
\end{aligned}$$

Au final, on obtient la hiérarchie représentée dans la figure suivante :



7.3. Analyse sémantique

Certaines expressions peuvent être syntaxiquement correctes mais ne pas avoir de sens par rapport à la sémantique du langage. Par exemple, en Pseudo Pascal, `new array of integer [4] + 2` est correct syntaxiquement mais n'a pas de sémantique. L'analyse sémantique étudie l'arbre de syntaxe abstraite produit par l'analyse syntaxique pour éliminer au maximum les programmes qui ne sont pas corrects du point de vue de la sémantique. Par exemple, on vérifie la portée des variables et le typage des expressions.

L'analyse sémantique est statique, c'est-à-dire qu'elle se fait au moment de la compilation. Par conséquent, elle ne peut exclure certains programmes dont on ne peut savoir s'ils sont corrects sémantiquement que dynamiquement, au moment de l'exécution. Par exemple, l'analyse sémantique ne rejettera pas un programme comme

```
var t : array of integer;
begin
  t := new array of integer[2];
  t[readline()] := 3
end
```

bien qu'il puisse ne pas avoir de sémantique si l'utilisateur rentre un entier plus grand que 2.

7.3.1. Table des symboles

Le fonctionnement typique de l'analyse sémantique est de parcourir l'AST pour vérifier certaines propriétés comme la portée des identificateurs ou le typage. Pour ne pas avoir à remonter dans l'arbre pour rechercher des informations sur la portée ou le type d'un

identificateur, on va stocker ces informations dans une structure externe appelée table des symboles, qui sera conservée tout au long du compilateur. Cette table sera créée dès l'analyse syntaxique et sera enrichie au cours des différentes phases du compilateur. Cette table associe à chaque identificateur du programme un ensemble de propriétés, dont la nature (nom de fonction, de variable, etc.), la portée, le type, etc. Elle peut être implémentée par exemple à l'aide d'une table de hachage.

7.3.2. Typage

Le typage permet de restreindre fortement le nombre de programme sémantiquement incorrects. On parle ici du typage statique, effectué au moment de la compilation, à ne pas confondre avec le typage dynamique qui peut avoir lieu au moment de l'exécution (comme en PHP, perl ou python). Le but est de s'assurer au moment de la compilation que le programme sera correct du point de vue des types au moment de l'exécution. L'analyse des types permet également de disposer d'informations utiles pour le compilateur, comme par exemple la taille des données en mémoire.

Le typage statique est en général obligé de rejeter des programmes qui sont pourtant correct sémantiquement, comme par exemple

```
var i : integer
if true then i := 0 else i:= 3 + new array of integer [2]
```

ce que ne ferait pas un typage dynamique, la deuxième branche n'étant jamais effectuée. Le typage statique est donc plus restrictif que le typage dynamique. Toutefois, en typage statique, une fois l'analyse de type effectuée, les types peuvent être complètement retirés du programme, tandis qu'en typage dynamique, les données doivent garder une information de typage, ce qui accroît leur taille en mémoire.

On distingue deux familles de typage statique : la simple vérification de type, comme en C ou en Pseudo Pascal, et l'inférence de type, comme en OCaml. Dans le premier cas, toutes les variables doivent être associés à un type, et le prototype des fonctions est explicite. Ce type est associé à l'identificateur dans la table des symboles au moment de l'analyse de la déclaration de variable ou de fonction. L'analyse de type se contente de parcourir l'AST de bas en haut en vérifiant que les types des arguments des fonctions et des opérateurs sont corrects (en utilisant la table des symboles pour trouver le types des variables et des fonctions).

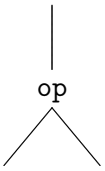
Pour l'inférence de type, le type des identificateurs et des fonctions est synthétisés automatiquement, sans que l'utilisateur ait à le spécifier explicitement (s'il le fait, l'analyse de type vérifie juste que le type donné par l'utilisateur est égal au type synthétisé lors de l'inférence). Dans le cadre de systèmes de types polymorphes ou avec sous-typage, on infère en fait le type le plus général, et on vérifie pour les applications que les arguments passés ont un type plus spécifique que le type attendu. L'inférence de type est bien entendu plus compliquée que la simple vérification de type (elle peut même devenir indécidable pour certains systèmes de types). L'algorithme utilisé pour l'inférence de type dans OCaml est l'algorithme de Hindley-Millner. Alors que la vérification de type est en

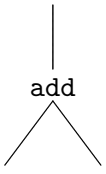
général linéaire, cet algorithme est PSPACE-complet (bien qu'en pratique l'inférence de type soit quasi linéaire).

8. Sélection d'instructions

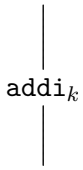
Pour passer du langage source au langage cible, on a besoin de faire correspondre les expressions de base du langage source à du code en langage cible. Souvent, cela ne constitue pas un problème majeur car les opérateurs du langage cible forment souvent un sur-ensemble de ceux du langage source. Par exemple, l'affectation `a := b;` peut être traduite par `mv` r_j`,` r_i, mais aussi par `addi` r_j`,` r_j`, 0` `andi` r_j`,` r_j`, 0xFFFF` `slli` r_j`,` r_j`, 0`... En fonction du contexte, certaines traductions sont préférable à d'autres, en général car elles produisent un code plus rapide, mais d'autres critères peuvent entrer en compte (consommation électrique, taille du code, etc.). De plus, un certain nombre de contraintes spécifiques à l'architecture cible sont à prendre en compte : certains registres sont dédiés aux opérations sur les entiers, ou les flottants ; certaines opérations prennent plus de temps que d'autres ; certaines opérations peuvent être pipelinées ; etc. La sélection d'instructions apparaît donc au niveau du backend. Néanmoins, si on veut avoir différentes architectures cibles sans avoir à réécrire la majeur partie du compilateur, on va utiliser une approche systématique de la sélection d'instructions.

8.1. Représentation intermédiaire (Untyped Pseudo-Pascal)

Pour cela, on va d'abord définir une représentation intermédiaire, qui reste sous forme d'arbre comme pour la syntaxe abstraite du langage source, mais qui utilise les opérateurs du langage cible. Pour chaque instruction RISC-V de la forme `op rd, rs, rt` on va créer un opérateur de la forme 

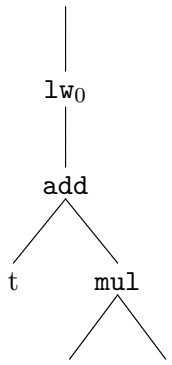
. Par exemple, on aura un opérateur binaire 

, mais puisqu'il existe aussi une instruction `addi` en RISC-V, on aura une famille

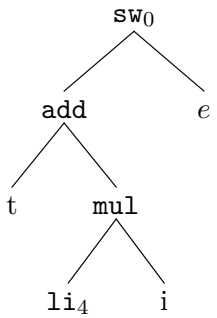


d'opérateurs unaires qui ajoute une constante k, pour tout entier machine k. La raison pour laquelle on conserve pour l'instant la forme arborescente est qu'il y sera plus facile d'optimiser la sélection d'instructions, en particulier grâce à la technique de réécriture : on traduit les arbres de syntaxe abstraite de façon naïve, puis on les réécrit pour obtenir des formes optimisées.

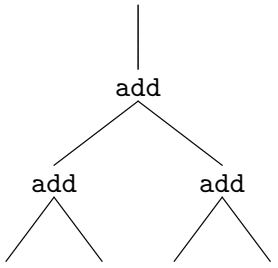
Pour chaque opérateur de Pseudo-Pascal, on choisit un opérateur, ou une succession d'opérateur RISC-V avec la même sémantique. Le choix est la plupart du temps trivial puisque les opérateurs de Pseudo-Pascal ont quasiment tous une version en RISC-V, par exemple `plus` et `add`, ou encore `<=` et `sle`. Seuls le moins unaire et les accès et écritures dans les tableau présentent une difficulté. Le premier peut être traduit comme `sub(li`$_0$`,x)`. Pour les tableaux, le problème provient du fait qu'en RISC-V on ne peut



accéder qu'à des zones mémoires. `t[i]` sera donc traduit par pour la lecture



et pour l'écriture d'une expression e. On devrait utiliser le typage de `t` pour savoir la taille des cases du tableaux. Comme en Pseudo-Pascal on n'a que des `integer` ou des tableaux (... de tableaux ...) d'`integer`, cette taille est forcément 4. Dans un langage plus évolué, elle peut varier. L'information de typage aura été stockée dans la table des symboles au cours de l'analyse sémantique.



Exemple 8.1 *:* On peut traduire naïvement `(x+2)+(4+y)` par $\texttt{x}\ \ \texttt{li}_2\ \ \texttt{li}_4\ \ \texttt{y}$, ce que l'on peut noter $\texttt{add(add(x,li}_2\texttt{),add(li}_4\texttt{,y))}$. Après optimisation, on souhaite obtenir par exemple $\texttt{addi}_6\texttt{(add(x,y))}$.

8.2. Réécriture

Définition 8.1 (Système de réécriture)**.** *Un* terme *est un arbre de syntaxe dans lequel apparaissent des* (méta-)variables *: ce sont de points de l'arbre qui peuvent être substitué par d'autres termes.*

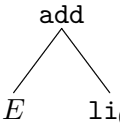
Une substitution σ *est une fonction de l'ensemble des variables V dans celui des termes $T(V)$ de support fini, c'est-à-dire que l'ensemble $\{X \in V \mid \sigma(X) \neq X\}$ est fini.*

Appliquer une substitution σ à un terme t revient à remplacer les occurrences des variables de t par leur image par σ.

Une règle de réécriture *est un couple formé de deux termes g et d tels que l'ensemble des variables de d est inclus dans celui de g. On la note $g \to d$.*

Un terme s peut se réécrire en t par la règle de réécriture $g \to d$ à la position $\mathfrak{p}$ et avec la substitution σ si $\sigma(g)$ est égal au sous-arbre de s à la position $\mathfrak{p}$ et t est le terme s dans lequel on a remplacé le sous-arbre à la position $\mathfrak{p}$ par $\sigma(d)$.

Un système de réécriture $\mathcal{R}$ *est un ensemble de règles de réécriture. On dit que s se réécrit en t par le système de réécriture $\mathcal{R}$ ($s \xrightarrow[\mathcal{R}]{} t$) si il existe une règle $g \to d$ dans $\mathcal{R}$, une substitution σ et une position $\mathfrak{p}$ de s telles que s se réécrit en t par la règle $g \to d$ à la position $\mathfrak{p}$ avec la substitution σ. On considère également la fermeture réflexive et transitive de cette relation (notée $\xrightarrow[\mathcal{R}]{*}$).*



Exemple 8.2 *:* Si E est une (méta-)variable, $\texttt{add}(E, \texttt{li}_0)$ est un terme. On le notera plus simplement via une syntaxe pseudo-concrète $\texttt{add}(E, \texttt{li}_0)$.

On peut considérer la règle de réécriture $\texttt{add}(E, \texttt{li}_0) \to E$. Le terme $\texttt{sub}(\texttt{add}(y, \texttt{add}(\texttt{mul}(\texttt{li}_2, z), \texttt{li}_0)), \texttt{li}_1)$ se réécrit alors en $\texttt{sub}(\texttt{add}(y, \texttt{mul}(\texttt{li}_2, z)), \texttt{li}_1)$ à la position 1.2 avec la substitution $\{x \mapsto \texttt{mul}(\texttt{li}_2, z)\}$.

Exemple 8.3 *:* Pour notre compilateur, on peut par exemple utiliser le système de réécri-

ture suivant pour les additions :

$$\texttt{add}(\texttt{li}_{k_1}, \texttt{li}_{k_2}) \to \texttt{li}_{k_1+k_2} \tag{8.1}$$
$$\texttt{add}(\texttt{li}_k, E) \to \texttt{addi}_k(E) \tag{8.2}$$
$$\texttt{add}(E, \texttt{li}_k) \to \texttt{addi}_k(E) \tag{8.3}$$
$$\texttt{add}(\texttt{li}_0, E) \to E \tag{8.4}$$
$$\texttt{add}(E, \texttt{li}_0) \to E \tag{8.5}$$
$$\texttt{addi}_{k_2}(\texttt{addi}_{k_1}(E)) \to \texttt{addi}_{k_1+k_2}(E) \tag{8.6}$$
$$\texttt{add}(\texttt{addi}_k(E_1), E_2) \to \texttt{addi}_k(\texttt{add}(E_1, E_2)) \tag{8.7}$$
$$\texttt{add}(E_1, \texttt{addi}_k(E_2)) \to \texttt{addi}_k(\texttt{add}(E_1, E_2)) \tag{8.8}$$

Dans ce système, `add(add(x,li(2)),add(li(4),y))` peut être réécrit de la façon suivante :

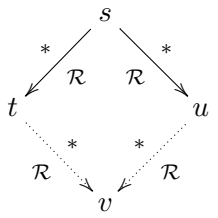
$$\begin{aligned}
\texttt{add(add(x,li(2)),add(li(4),y))} &\longrightarrow \texttt{add(addi(x,2),add(li(4),y))} \\
&\longrightarrow \texttt{add(addi(x,2),addi(y,4))} \\
&\longrightarrow \texttt{addi(add(x,addi(y,4)),2)} \\
&\longrightarrow \texttt{addi(addi(add(x,y),4),2)} \\
&\longrightarrow \texttt{addi(add(x,y),6)}
\end{aligned}$$

Il est possible d'utiliser n'importe quel système de réécriture pour optimiser la traduction naïve, à condition que :
- chacune des règles préserve la sémantique du langage ;
- l'application du système de réécriture termine quelque soit l'ordre dans lequel on applique les règles ; on dit alors que le système de réécriture est fortement normalisant ;
- l'ordre d'application des règles est indifférent au final ; on dit alors que le système est confluent.

Ces deux dernières conditions garantissent qu'il existera une et une seule forme optimisée de la traduction naïve.

Définition 8.2 (Terminaison, confluence)**.** *Un système de réécriture $\mathcal{R}$ est dit* fortement normalisant*, ou encore* terminant*, si la relation $\xrightarrow[\mathcal{R}]{}$ est bien fondée, c'est-à-dire s'il n'existe pas de suite infinie de réécriture $s_1 \xrightarrow[\mathcal{R}]{} s_2 \xrightarrow[\mathcal{R}]{} \cdots \xrightarrow[\mathcal{R}]{} s_n \xrightarrow[\mathcal{R}]{} \cdots$.*

Un système de réécriture est confluent *si pour tous termes s, t, u, si $s \xrightarrow[\mathcal{R}]{*} t$ et $s \xrightarrow[\mathcal{R}]{*} u$ alors il existe un v tel que $t \xrightarrow[\mathcal{R}]{*} v$ et $u \xrightarrow[\mathcal{R}]{*} v$. Graphiquement, cela donne :*



Le but est d'obtenir une forme canonique :

Définition 8.3 (Forme canonique)**.** *Un terme s est en* forme canonique *par rapport un système de réécriture $\mathcal{R}$ si il n'existe aucun t tel que $s \xrightarrow[\mathcal{R}]{} t$.*

Proposition 8.1. *Si $\mathcal{R}$ est confluent et terminant, alors tout terme possède une et une seule forme canonique.*

Une technique pour prouver la terminaison d'un système de réécriture est d'utiliser l'ordre de Knuth et Bendix.

Définition 8.4 (Ordre de Knuth et Bendix)**.** *Soit un ordre $\succ$ total sur les symboles de fonction, appelé précédence. On note $|t|$ la taille d'un terme t.*

*On définit l'*ordre de Knuth et Bendix $>_{KBO}$ *par $s >_{KBO} t$ si :*

- *pour chaque variable X, le nombre d'occurrence de X dans t est inférieur à celui dans s ;*
- *et*
  - *soit $|s| > |t|$ (pour l'ordre usuel sur les entiers) ;*
  - *soit $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$ et $f \succ g$;*
  - *soit $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$ et il existe i tel que pour tout $j < i$ on a $s_j = t_j$ et $s_i >_{KBO} t_i$.*

Proposition 8.2. *S'il existe une précédence $\succ$ tel que pour chaque règle $g \to d$ de $\mathcal{R}$ on a $g >_{KBO} d$, alors $\mathcal{R}$ termine.*

Exemple 8.4 *:* Si on prend comme précédence $\texttt{add} \succ \texttt{addi}_k$ pour tout k alors le système de l'exemple 8.3 termine : pour les six premières règles, la taille du terme diminue ; pour les deux dernière, la taille est la même mais la précédence permet de conclure.

Remarque 8.1 *:* Il existe des systèmes dont la terminaison ne peut pas être montrée par l'utilisation de l'ordre de Knuth et Bendix. (Par exemple si certaines règles font croître la taille des termes, ou dupliquent des variables.) Néanmoins, il existe aujourd'hui des outils automatiques capables de la démontrer dans de nombreux cas (exemples : AProVE [1], ou CiME [2], ce dernier permettant également de prouver la confluence).

Une fois qu'on a montré la terminaison, on peut montrer la confluence en montrant la confluence locale.

Définition 8.5 (Paire critique)**.** *Étant donné un système de réécriture $\mathcal{R}$, s'il existe deux règles de réécriture (pas forcément différentes) $g \to d$ et $l \to r$ dans $\mathcal{R}$, une position $\mathfrak{p}$ dans g (différente de la racine si on considère deux fois la même règle) telle que $g|_\mathfrak{p}$ n'est pas une variable et $g|_\mathfrak{p}$ et l sont unifiables (c'est-à-dire qu'il existe une substitution σ telle que $\sigma(g|_\mathfrak{p}) = \sigma(l)$), alors le couple de termes $\sigma(d), \sigma(g[r]_\mathfrak{p})$ est appelée une* paire critique *de $\mathcal{R}$ (pour σ l'unificateur le plus général de $g|_\mathfrak{p}$ et l).*

Une paire critique s, t est joignable *s'il existe un terme v tel que $s \xrightarrow{*} v \xleftarrow{*} t$.*

1. http://aprove.informatik.rwth-aachen.de/
2. http://cime.lri.fr/

Proposition 8.3. *Si un système est terminant et si toutes ses paires critiques sont joignables, alors il est confluent.*

Étant donné un système de réécriture terminant, il convient donc pour vérifier sa confluence de regarder les règles de réécriture deux à deux pour vérifier que leurs paires critiques éventuelles sont joignables.

Exemple 8.5 *:* À partir de l'exemple 8.3, on peut regarder quels sont les paires critiques :

— (4.1) et (4.1) : pas de paire critique
— (4.1) et (4.2) : $\texttt{li}_{k_1+k_2}$, $\texttt{addi}_{k_2}(\texttt{li}_{k_1})$
— (4.1) et (4.3) : $\texttt{li}_{k_1+k_2}$, $\texttt{addi}_{k_1}(\texttt{li}_{k_2})$
— (4.1) et (4.4) : $\texttt{li}_{k_2}$, $\texttt{li}_{k_2}$
— (4.1) et (4.5) : $\texttt{li}_{k_1}$, $\texttt{li}_{k_1}$
— (4.1) et (4.6) : pas de paire critique
— (4.1) et (4.7) : pas de paire critique
— (4.1) et (4.8) : pas de paire critique
— (4.2) et (4.2) : pas de paire critique
— (4.2) et (4.3) : $\texttt{addi}_{k_1}(\texttt{li}_{k_2})$, $\texttt{addi}_{k_2}(\texttt{li}_{k_1})$
— (4.2) et (4.4) : $\texttt{addi}(E, 0)$, E
— (4.2) et (4.5) : $\texttt{addi}_k(\texttt{li}(0))$, $\texttt{li}_k$
— (4.2) et (4.6) : pas de paire critique
— (4.2) et (4.7) : pas de paire critique
— (4.2) et (4.8) : $\texttt{addi}_{k_1}(\texttt{addi}_{k_2}(E))$, $\texttt{addi}_{k_2}(\texttt{add}(\texttt{li}_{k_1}, E))$
— (4.3) et (4.3) : pas de paire critique
— (4.3) et (4.4) : $\texttt{addi}_k(\texttt{li}(0))$, $\texttt{li}_k$
— (4.3) et (4.5) : $\texttt{addi}(E, 0)$, E
— (4.3) et (4.6) : pas de paire critique
— (4.3) et (4.7) : $\texttt{addi}_{k_2}(\texttt{addi}_{k_1}(E))$, $\texttt{addi}_{k_1}(\texttt{add}(E, \texttt{li}_{k_2}))$
— (4.3) et (4.8) : pas de paire critique
— (4.4) et (4.4) : pas de paire critique
— (4.4) et (4.5) : $\texttt{li}(0)$, $\texttt{li}(0)$
— (4.4) et (4.6) : pas de paire critique
— (4.4) et (4.7) : pas de paire critique
— (4.4) et (4.8) : $\texttt{addi}_k(E)$, $\texttt{addi}_k(\texttt{add}(\texttt{li}(0), E))$
— (4.5) et (4.5) : pas de paire critique
— (4.5) et (4.6) : pas de paire critique
— (4.5) et (4.7) : $\texttt{addi}_k(E)$, $\texttt{addi}_k(\texttt{add}(E, \texttt{li}(0)))$
— (4.5) et (4.8) : pas de paire critique
— (4.6) et (4.6) : $\texttt{addi}_{k_1}(\texttt{addi}_{k_2+k_3}(E))$, $\texttt{addi}_{k_1+k_2}(\texttt{addi}_{k_3}(E))$
— (4.6) et (4.7) : $\texttt{addi}_{k_2}(\texttt{add}(\texttt{addi}_{k_1}(E_1), E_2))$, $\texttt{add}(\texttt{addi}_{k_1+k_2}(E_1), E_2)$
— (4.6) et (4.8) : $\texttt{addi}_{k_1}(\texttt{add}(E_1, \texttt{addi}_{k_2}(E_2)))$, $\texttt{add}(E_1, \texttt{addi}_{k_1+k_2}(E_2))$
— (4.7) et (4.7) : pas de paire critique
— (4.7) et (4.8) : $\texttt{addi}_{k_1}(\texttt{add}(E_1, \texttt{addi}_{k_2}(E_2)))$, $\texttt{addi}_{k_2}(\texttt{add}(\texttt{addi}_{k_1}(E_1), E_2))$
— (4.8) et (4.8) : pas de paire critique

On regarde ensuite si ces paires critiques sont joignables. Par exemple, celle entre (4.3)

et (4.7) est joignable :

$$\texttt{addi}_{k_2}(\texttt{addi}_{k_1}(E)) \xrightarrow{*} \texttt{addi}_{k_1+k_2}(E) \xleftarrow{*} \texttt{addi}_{k_1}(\texttt{add}(\texttt{li}_{k_2}, E))$$

Seules celles entre (4.1) et (4.2), (4.1) et (4.3), (4.2) et (4.3), (4.2) et (4.4), (4.2) et (4.5), (4.3) et (4.4) et enfin (4.3) et (4.5) ne sont pas joignables. Le système n'est donc pas confluent. Par contre, on s'aperçoit qu'il est possible de rajouter de nouvelles règles pour permettre de fermer les paires critiques non joignables :

$$\texttt{addi}_{k_2}(\texttt{li}_{k_1}) \to \texttt{li}_{k_1+k_2} \tag{8.9}$$

$$\texttt{addi}_0(E) \to E \tag{8.10}$$

Une fois ces règles ajoutées, on vérifie que le système est toujours terminant et que les paires critiques nouvellement créées par les règles ajoutées sont joignables. Il s'agit de

— (4.9) et (4.6) : $\texttt{addi}_{k_3}(\texttt{li}_{k_1+k_2})$, $\texttt{addi}_{k_2+k_3}(\texttt{li}_{k_1})$
— (4.9) et (4.7) : $\texttt{add}(\texttt{li}_{k_1+k_2}, E_2)$, $\texttt{addi}_{k_2}(\texttt{add}(\texttt{li}_{k_1}, E_2))$
— (4.9) et (4.8) : $\texttt{add}(E_1, \texttt{li}_{k_1+k_2})$, $\texttt{addi}_{k_2}(\texttt{add}(E_1, \texttt{li}_{k_1}))$
— (4.9) et (4.10) : $\texttt{li}_{k_1}$, $\texttt{li}_{k_1}$
— (4.10) et (4.6) : $\texttt{addi}_{k_2}(E)$, $\texttt{addi}_{k_2}(E)$
— (4.10) et (4.7) : $\texttt{add}(E_1, E_2)$, $\texttt{addi}(\texttt{add}(E_1, E_2), 0)$
— (4.10) et (4.8) : $\texttt{add}(E_1, E_2)$, $\texttt{addi}(\texttt{add}(E_1, E_2), 0)$

qui sont effectivement joignables. Le système complété est donc confluent.

Plus généralement, il existe une procédure qui, étant donné un système de réécriture, le transforme afin de le rendre confluent et terminant : c'est la procédure de complétion de Knuth-Bendix. La terminaison et la confluence d'un système de réécriture étant indécidables, cette procédure peut ne pas terminer ou échouer.

Exemple 8.6 *:* On pourrait rajouter une règle

$$\texttt{add}(\texttt{sub}(\texttt{li}(0), x), y) \to \texttt{sub}(y, x)$$

Néanmoins cela ne serait pas correct car cette règle ne préserve pas la sémantique du Pseudo-Pascal. En effet, elle change l'ordre d'exécution de x et y. Par exemple, si la fonction `g` modifie la variable `x`, `(0 - g()) + x` n'a pas la même sémantique que `x - g()`. Pour quand même appliquer cette règle dans les cas où cela est possible, on la restreint aux cas où x et y sont pures. Une expression est dite pure si elle n'effectue aucune écriture dans une variable ou dans un tableau. Cette condition est nécessaire pour s'assurer que la sémantique est préservée lors de l'application de la dernière règle. Pour décider si une expression est pure, il faut vérifier que sa sémantique ne change pas l'environnement.

Il est possible d'utiliser une condition moins stricte dans le dernier exemple qui dit qu'il est possible de partager l'environnement en deux parties E_1 et E_2 telles que la sémantique de e_1 ne change pas l'environnement E_2 et la sémantique de e_2 ne modifie pas E_1. Ceci permet par exemple de faire commuter `(0 - g()) + x` en `x - g()` si `g` ne modifie pas `x`. Néanmoins, dans le cas d'un langage à pointeur comme le C, cette analyse s'avère impossible.

8.2.1. Implémentation

La traduction qu'on veut obtenir est donc la forme canonique de la traduction naïve du programme source. Pour obtenir une implémentation simple et efficace, plutôt que de construire la traduction naïve puis de la normaliser, on va normaliser au fur et à mesure de la construction.

Pour cela, on va écrire des fonctions qui, étant donnés des expressions filles déjà en forme canonique, leur applique un constructeur et réduisent aussitôt l'expression ainsi obtenue en forme canonique. On appelle parfois ces fonctions *smart constructors*.

Par exemple, on aura deux fonctions `add` et `addi` permettant de construire respectivement des nœuds d'addition à partir de deux expressions supposées en forme normale et des nœuds d'addition d'une expression avec une constante. Ensuite, on utilisera ces fonctions plutôt que les constructeurs `Add` et `Addi`.

Étant donné le système de réécriture de l'exemple 8.3, complété dans les exemples 8.5 et 8.6, ces fonctions pourront s'écrire on OCaml :

```
let rec addi k e =
  match k, e with
      0, _ -> e
    | k2, Addi(k1, e) -> addi (k1+k2) e
    | k2, Li(k1) -> Li (k1+k2)
    | _ -> Addi(k, e)

let rec add e1 e2 =
  match e1, e2 with
    Li(0), e -> e
  | e, Li(0) -> e
  | Li(k1), Li(k2) -> Li (k1+k2)
  | Li(k), e -> addi e k
  | e, Li(k) -> addi e k
  | Addi(k, e1), e2 -> addi k (add e1 e2)
  | e1, Addi(k, e2) -> addi k (add e1 e2)
  | Sub(Li(0), x), y when pure x && pure y -> sub y x
  | _ -> Add(e1, e2)

and sub e1 e2 =
  ...
```

Noter comment les appels récursifs permettent de poursuivre la réécriture. Puisque le système est confluent, l'ordre des branches n'est pas important. Néanmoins on peut s'arranger pour minimiser le nombre d'appels récursifs.

Remarque 8.2 *:* Pour être certain ensuite de n'employer que les *smart constructors* pour construire des expressions, et ainsi s'assurer par construction de ne manipuler que des objets en forme normale, on peut utiliser les types sommes privés de OCaml.

9. Graphe de flot de contrôle

La forme arborescente utilisée jusqu'ici n'est pas satisfaisante car elle ne permet pas de voir la séquentialité des actions. Nous allons donc la transformer en un graphe dont les nœuds seront des instructions élémentaires. Un nœud sera relié aux instructions le suivant lors de l'exécution. Les instructions élémentaires seront les instructions du langage cible. Pour rester suffisamment abstrait, on continuera toutefois à utiliser un nombre illimité de variables, qu'on verra plutôt comme des pseudo-registres. Chaque pseudo-registre sera local à la fonction où il est utilisé, et donc préservé lors des appels.

Une telle représentation est intéressante car :

— l'organisation en graphe facilite l'insertion et la suppression d'instructions dans les phases d'optimisations ;
— elle est simple et générale, et peut donc refléter les constructions usuelles `if`, `while`, `for`, `break`, `continue`, et même `goto` ;
— la structure arborescente n'est plus utile au-delà ;
— le passage de pseudo-registres à registres réels peut être reporté plus tard (cf. le chapitre 11).

On notera les pseudo-registres `%`i où i est un entier positif.

9.1. Register Transfer Language

Le graphe de flot de contrôle est donné à l'aide du *Register Transfer Language*. Celui-ci est composé d'instructions de base de la forme `label1: op %i, %j, k -> label2, label3` où

— `label1` désigne le label d'entrée ;
— `label2` et `label3` sont des labels de sortie ;
— `op` est un opérateur du langage cible, à ceci près que les appels de fonctions ne sont pas encore explicités et sont donc de la forme `l1: call %i, f(%j,%k)` ;
— le premier argument de `op` est un pseudo-registre dans lequel est stocké le résultat ;
— les autres arguments sont des pseudo-registres ou des constantes (sauf dans le cas de `call` une nouvelle fois).

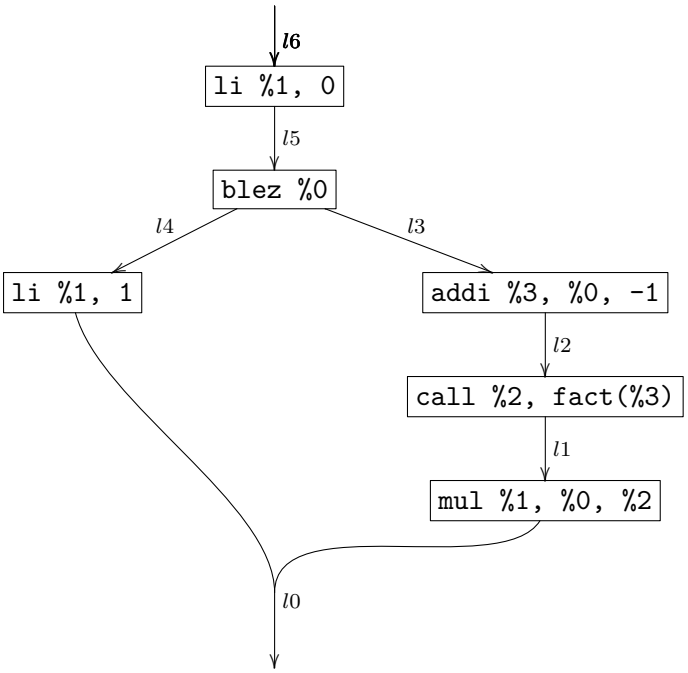
Chaque instruction de base produit un nœud. On le relie aux nœuds correspondant à ses labels de sortie.

Pour chaque fonction, on indique les pseudo-registres contenant les arguments ainsi que le pseudo-registre dans lequel sera placé le résultat de la fonction. Un label d'entrée et de sortie de la fonction est également précisé, ainsi que les pseudo-registres utilisés dans le corps de la fonction.

Exemple 9.1 *:* Voici une version possible de la factorielle en RTL :

```
function fact(%0) : %1
var %0, %1, %2, %3
entry l6
exit l0
l6: li %1, 0  -> l5
l5: blez %0  -> l4, l3
l3: addi %3, %0, -1  -> l2
l2: call %2, fact(%3)  -> l1
l1: mul %1, %0, %2  -> l0
l4: li %1, 1  -> l0
```

Ce qui correspond à un graphe



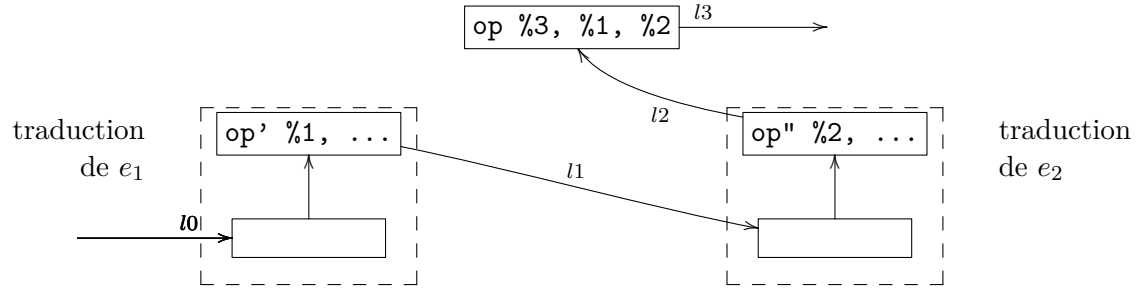
9.2. Calcul du graphe

Pour calculer le graphe de flot de contrôle, on part de l'arbre de syntaxe obtenu après sélection d'instructions, et on effectue un parcours en profondeur.

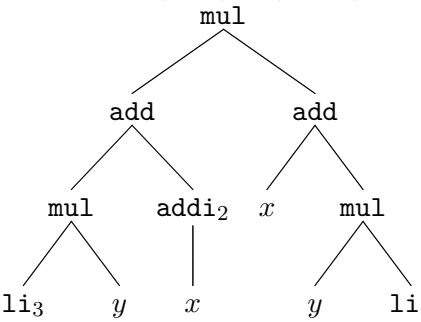
On traduit les expressions, les conditions et les instructions.

Pour les expressions, chaque nœud de l'AST correspondra à un noeud du graphe. Le résultat de chaque sous-expression sera mis dans un pseudo-registre frais, qui sera utilisé dans les expressions utilisant cette sous-expression, et un environnement sera utilisé pour mémoriser le lien entre variables et pseudo-registres. Les fragments de graphe correspondant aux différentes sous-expressions seront reliés les uns aux autres de manière à respecter l'ordre d'évaluation imposé par la sémantique.

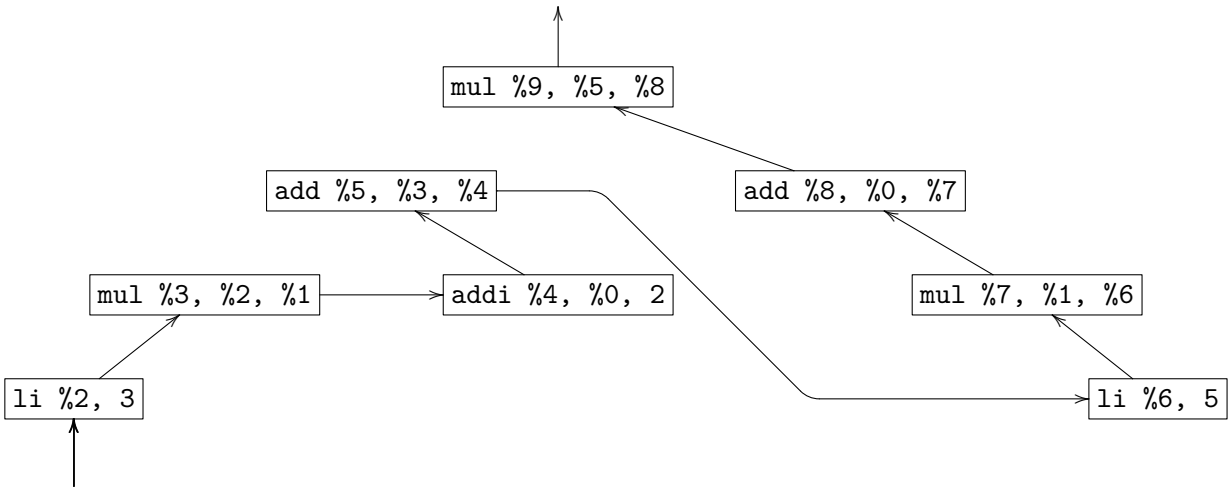
Ainsi, pour une expression UPP `op`(e_1, e_2) à partir d'un nœud d'entrée `l0`, on va traduire récursivement la sous-expression e_1 avec comme entrée `l0`, ce qui donnera un graphe dont le nœud de sortie est `l1` et qui met le résultat dans un pseudo-registre `%1`. On traduit ensuite la sous-expression e_2 avec comme label d'entrée `l1`, ce qui donne un graphe dont le nœud de sortie est `l2` et qui met le résultat dans un pseudo-registre `%2`. On ajoute ensuite un nœud `l2: op %3, %1, %2 -> l3` pour un pseudo-registre `%3` et un label de sortie `l3` frais tous les deux.



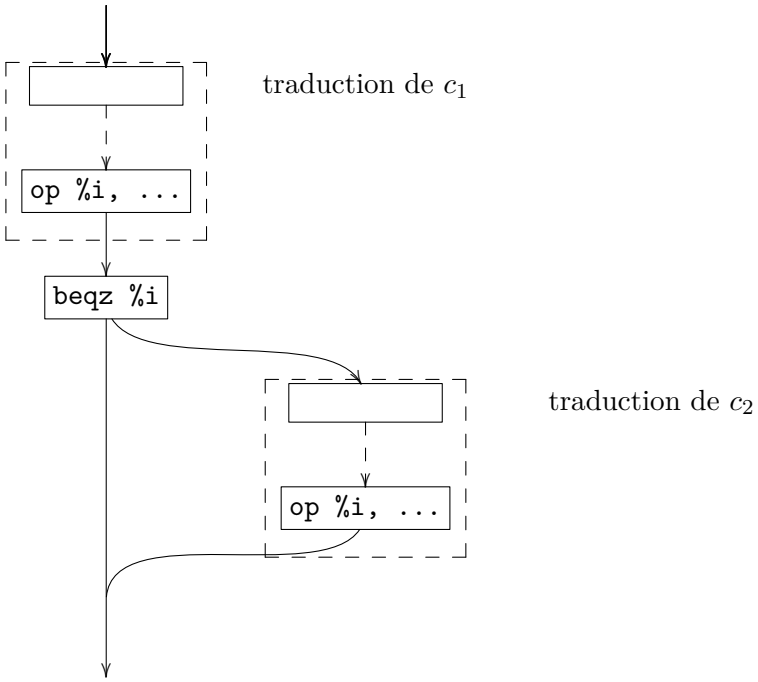
Exemple 9.2 *:* Si on considère l'expression Pseudo-Pascal `(3 * y + (2 + x)) * (x + y * 5)`, dont la traduction en UPP sera `mul`(`add`(`mul`($\texttt{li}_3, y$), $\texttt{addi}_2(x)$), `add`$(x,$ `mul`$(y, \texttt{li}_5))$),



ce qui correspond à l'arbre, en supposant que x est dans le pseudo-registre `%0` et y dans `%1`, on obtiendra le graphe suivant :



Pour la traduction des conditions, on va également mettre le résultat de l'évaluation de la condition dans un pseudo-registre frais. On procède comme ci-dessus en cas d'expression à valeur booléenne ou de `not`, mais il faut prendre en considération le comportement coupe-circuit des opérations logiques `and` et `or`. Ainsi, on devra faire un branchement suivant que l'évaluation de la sous-condition gauche est vrai ou faux, puisqu'on aura ou pas à évaluer la sous-condition droite. Pour $\texttt{and}(c_1, c_2)$ on aura donc :



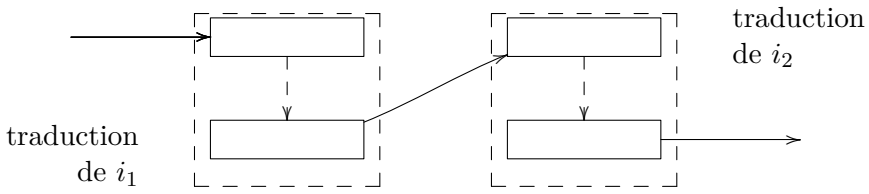
On prendra soin de bien utiliser le même pseudo-registre `%i` dans les deux branches (ce qu'on peut faire puisqu'on le prend frais).

Pour la traduction des instructions, on va distinguer les différents cas. Pour les affectations dans des variables `x :=` e, on va traduire l'expression e ; ce qui donnera un graphe dont le nœud avant la sortie utilise le pseudo-registre frais `%i` comme destination. Il suffit de remplacer ce pseudo-registre par celui correspondant à la variable `x`.

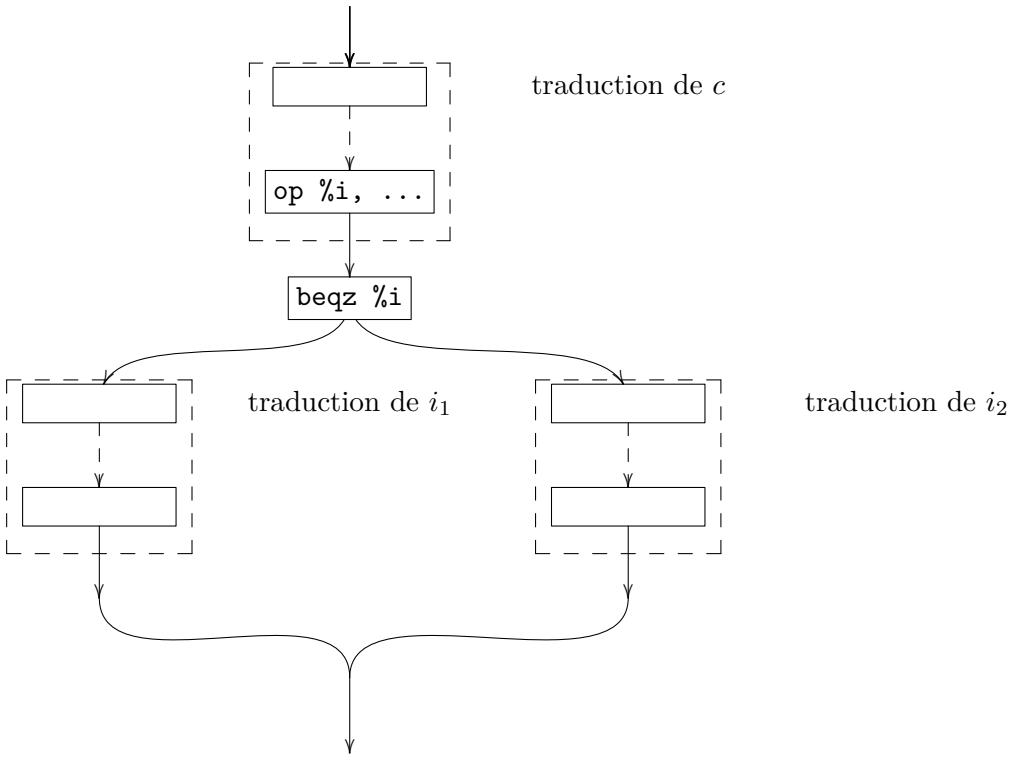
Pour l'affectation dans un tableau e_1`[`e_2`]` `:=` e_3, la traduction en UPP donne une traduction naïve $\texttt{sw}_0(e'_3, \texttt{add}(e'_1, \texttt{mul}_2(e'_2, \texttt{li}_4)))$ qui va être optimisée en un expression $\texttt{sw}_k(e''_3, e)$. Il suffit donc de traduire en graphe de flot de contrôle les sous-expressions e''_3 et e, qui mettent leur résultat respectivement dans les pseudo-registres frais `%i` et `%j` puis de rajouter un nœud `sw %i, k(%j)`.

La traduction d'une séquence d'instruction ce fait tout naturellement en joignant séquentiellement les graphes correspondant aux traductions des instructions. i_1`;` i_2 de-

vient :



La traduction des conditionnelles va introduire d'autres branchements dans le graphe. Dans sa forme la plus simple, la traduction d'une conditionnelle consiste à évaluer l'expression de la conditionnelle, qui est alors placée dans un registre contenant par conséquent 0 ou 1 ; puis à utiliser par exemple `beqz` ; traduire les deux branches, chacune étant reliée à une sortie de `beqz` ; et faire se rejoindre les deux branches à leur sortie (c'est-à-dire utiliser le même label de sortie). Ainsi, si on part de `if` c `then` i_1 `else` i_2 alors on a



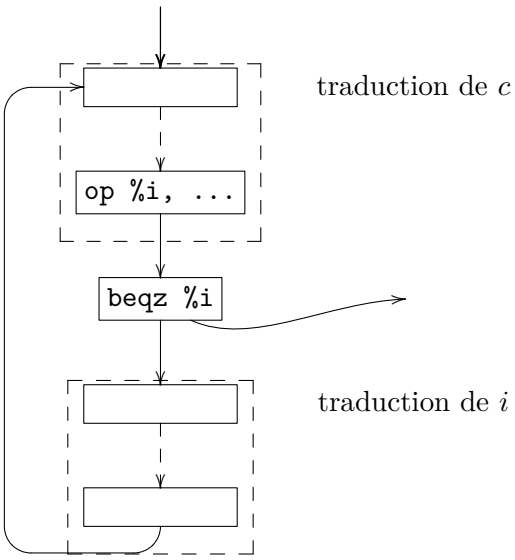
Il est possible d'optimiser certains calculs de conditionnelles, par exemple pour `x + y < 0` on fera

```
l1: add %2, %0, %1  -> l2
l2: bltz %2  -> l5, l4
```

au lieu de

```
l1: add %2, %0, %1  -> l2
l2: slti %3, %2, 0  -> l3
l3: beqz %3  -> l4, l5
```

Enfin, les boucles du programme sont naturellement traduites en cycles dans le graphe. Ainsi, pour traduire `while` c `do` i on fera



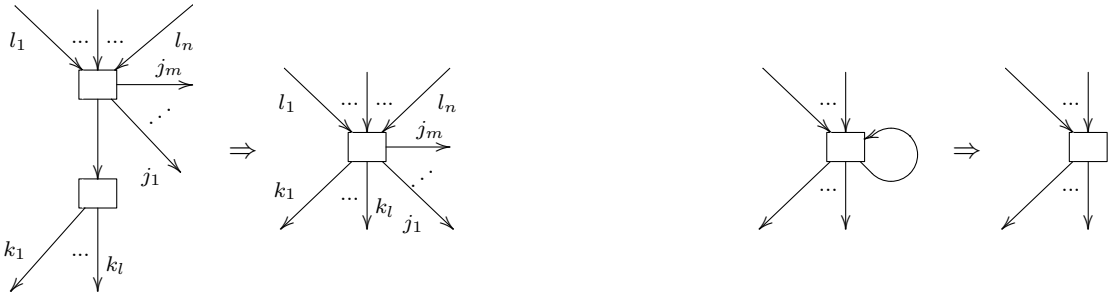
Le graphe de flot de contrôle est donc cyclique. Néanmoins, en l'absence de `goto`, le graphe reste réductible :

Définition 9.1 (Domination, Réductibilité)**.** *Un nœud m d'un graphe* domine *un nœud n si tout chemin du point d'entrée du graphe vers n passe par m.*

Un graphe est dit réductible *si dans tout cycle, il existe un sommet qui domine tous les autres.*

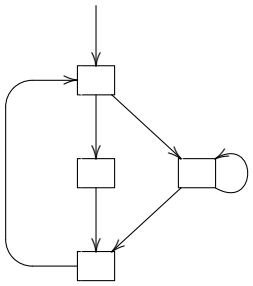
Intuitivement, dans un graphe réductible, chaque boucle n'admet qu'un seul point d'entrée. On ne peut pas sauter directement à l'intérieur. Il est possible de donner la caractérisation des graphes réductibles :

Proposition 9.1. *Un graphe est réductible si en appliquant les transformations suivantes sur le graphe on obtient un graphe composé d'un seul sommet :*

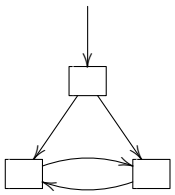


La première transformation fusionne un nœud avec son prédécesseur si celui-ci est unique, la seconde enlève les boucles d'un nœud vers lui-même.

Exemple 9.3 *:* Le graphe suivant est réductible :



tandis que



ne l'est pas.

De nombreux algorithme d'analyse du graphe de flot de contrôle sont plus efficaces sur les graphes réductibles (voire ils ne sont définis que pour eux), d'où l'intérêt d'avoir un langage de haut niveau ne permettant pas des branchements n'importe où.

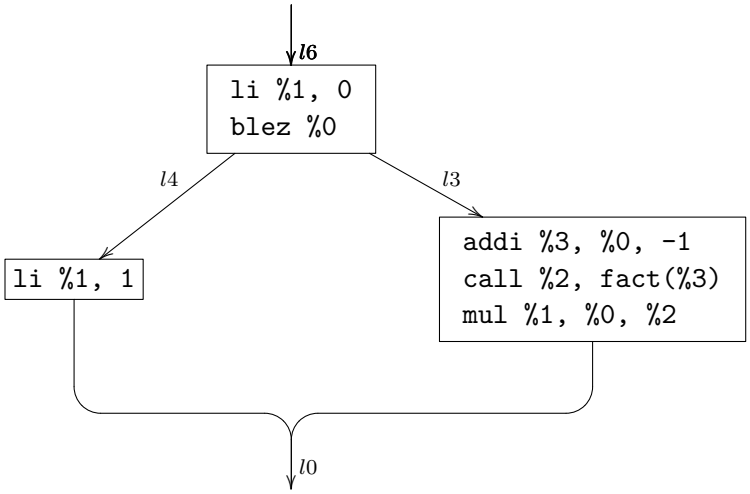
Remarque 9.1 *:* Si un graphe est réductible, le graphe obtenu en inversant les arêtes n'est pas forcément réductible. En particulier, dans le cas d'un langage avec boucles pouvant être interrompues par un `break`, le graphe inversé permettra deux entrées dans la boucle (celle correspondant à la sortie normale, et celle correspondant au `break`). Or, certains algorithmes, en particulier le calcul de la durée de vie (cf. section 11.1), travaillent en fait sur le graphe inversé.

Il peut être intéressant de regrouper les instructions de base en bloc :

Définition 9.2. *Un* bloc de base *est un ensemble d'instructions de base* $i_1, \ldots, i_n$ *tel pour tout* $1 \leq j < n$*, l'instruction* i_j *n'a qu'un seul successeur qui est* i_{j+1}*, et l'instruction* i_{j+1} *n'a qu'un prédécesseur (qui est par conséquent* i_j*).*

Un bloc de base contient donc des instructions telles que, pour en exécuter une, il faut les exécuter toutes. Le graphe avec des bloc de base contient moins de sommets, ce qui conduit à des algorithmes plus efficaces en pratique.

Exemple 9.4 *:* En regroupant les instructions de l'exemple 9.1, on obtient le graphe suivant :



9.3. Suppression des calculs redondants

Il arrive que dans un programme que certains calculs soient redondants, par exemple on peut avoir une boucle

```
while b do
  x := x + (2 * y);
```

Il vaut alors mieux faire le calcul de `2*y` en dehors de la boucle :

```
z := 2 * y;
while b do
  x := x + z;
```

Le passage à RTL peut ajouter d'autres redondances, qui ne peuvent pas être évitées en modifiant le code source.

Exemple 9.5 *:* En partant du programme

```
x := t[i];
t[i] := t[i-1];
t[i-1] := x;
```

les traductions optimisées de `t[i]` et `t[i-1]` en UPP seront respectivement $\texttt{lw}_0(t, \texttt{slli}_2(i))$ et $\texttt{lw}_{-4}(t, \texttt{slli}_2(i))$. En supposant que x, t et i sont respectivement dans les pseudo-registres `%1`, `%2` et `%3`, on obtient donc naïvement le code RTL (en omettant les labels comme le code est linéaire) :

```
1  slli %4, %3, 2
2  add %5, %2, %4
3  lw %1, 0(%5)
4  slli %6, %3, 2
5  add %7, %2, %6
6  slli %8, %3, 2
7  add %9, %2, %8
8  lw %10, -4(%9)
9  sw %10, 0(%7)
10 slli %11, %3, 2
11 add %12, %2, %11
12 sw %1, -4(%12)
```

On voit que `%2 + 4 * %3` est calculé quatre fois. Ce calcul est celui de l'adresse que l'on pourrait écrire, en C, de la forme `t + i`. Il n'y a aucun moyen de modifier le code source pour éviter cette redondance, qui doit donc être éliminée au niveau de RTL.

Le code que l'on aimerait obtenir est

```
slli %4, %3, 2
add %5, %2, %4
lw %1, 0(%5)
lw %11, -4(%5)
sw %11, 0(%5)
sw %1, -4(%5)
```

dans lequel on effectue le calcul de l'adresse qu'une seule fois, que l'on met dans `%5` puis que l'on utilise quatre fois.

Pour détecter les calculs redondants, il faut essayer de trouver des relations entre les pseudo-registres qui seront vérifiées à toute exécution du code. Pour cela, on va simuler l'exécution du code en affectant à chaque pseudo-registres et à chaque point du code une valeur symbolique.

Définition 9.3 (Valeur symbolique)**.** *Les* valeurs symboliques *sont construites par la syntaxe abstraite*

$$e ::= \alpha \mid \underline{k} \mid \mathtt{op}(e_1, \ldots, e_n)$$

où α est une variable symbolique (représentant une valeur inconnue), $\underline{k}$ est une constante et `op` *est un opérateur UPP.*

Au départ, on n'a aucune information sur le contenu des pseudo-registres, donc on associe à chaque pseudo-registre une variable symbolique différente. On simule ensuite l'exécution du code pour mettre à jour les valeurs symboliques des pseudo-registres.

Exemple 9.6 *:* Sur le code avec redondance de l'exemple 9.5 on obtient l'analyse donnée dans le tableau 9.1.

On voit que les instructions `lw` font intervenir des variables symboliques fraîches, puisqu'on ne connaît pas a priori le contenu de la mémoire. Il en serait de même avec `call`. On peut pousser l'analyse plus loin pour traiter ces cas, mais nous ne le feront pas ici.

Une fois l'analyse effectuée, on peut faire les optimisations en remplaçant les calculs redondants par des `mv` : si la valeur symbolique affectée dans un pseudo-registre existe déjà dans un autre pseudo-registre, on remplace l'instruction par un `mv` avec ce pseudo-registre.

Exemple 9.7 *:* Si on effectue cette opération sur le code de l'exemple 9.5 on obtient :

```
1  slli %4, %3, 2
2  add %5, %2, %4
3  lw %1, 0(%5)
4  mv %6, %4
5  mv %7, %5
6  mv %8, %4
7  mv %9, %5
8  lw %10, -4(%9)
9  sw %10, 0(%7)
10 mv %11, %4
11 mv %12, %5
12 sw %1, -4(%12)
```

L'intérêt d'une telle transformation peut sembler nul, puisqu'on obtient le même nombre d'instructions. Néanmoins, le code s'améliorera lors d'optimisations ultérieures :

— Les instructions des lignes 4, 6 et 10 seront supprimées lors de l'élimination du code mort (cf. section 11.1.1) (le pseudo-registre qu'elles affectent n'est jamais utilisé par la suite) ;

— les pseudo-registres `%5`, `%7`, `%9` et `%12` seront alloués si possible au même (véritable) registre au moment de l'allocation de registre (cf. section 11.2), auquel cas on pourra supprimer les instructions `mv`.

Tel qu'on l'a vu, le calcul des valeurs symboliques ne peut-être effectué qu'à l'intérieur d'un bloc de base, car dans une boucle, la valeur symbolique d'un pseudo-registre pourrait dépendre de sa valeur à l'itération précédente de la boucle. Il peut être alors utile de mettre le code en forme SSA :

Définition 9.4 (*Static Single Assignement)***.** *Un graphe de flot de contrôle est en forme SSA si :*

— *chaque pseudo-registre n'est affecté qu'une seule fois ;*

— *chaque utilisation d'un pseudo-registre est dominée par un sommet où il est affecté.*

Mettre un code linéaire en forme SSA n'est pas compliqué, il suffit de s'assurer que les pseudo-registres sont bien initialisés, puis utiliser des pseudo-registres frais à chaque affectation. Par contre, quand il y a des boucles, on ne peut pas réaffecter un pseudo-registre déjà affecté. Par conséquent, on utilise des ϕ-fonctions qui permettre de joindre

l.	%1	%2	%3	%4	%5	%6	%7	%8	%9	%10	%11	%12
	α	β	γ									
1	α	β	γ	$\texttt{slli}_2(\gamma)$								
2	α	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$							
3	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$							
4	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$						
5	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$					
6	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$				
7	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$			
8	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	ϵ		
9	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	ϵ		
10	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	ϵ	$\texttt{slli}_2(\gamma)$	
11	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	ϵ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$
12	δ	β	γ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$	ϵ	$\texttt{slli}_2(\gamma)$	$\texttt{add}(\beta, \texttt{slli}_2(\gamma))$

TABLE 9.1. – Analyse de redondance de l'exemple 9.5

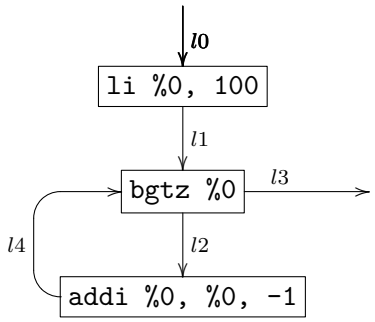
les valeurs des fonctions de deux pseudo-registres. Nous n'irons pas plus loin à propos de la forme SSA que de présenter les exemples suivants. Le lecteur intéressé pourra lire Appel [1998].

Exemple 9.8 *:* Le programme de l'exemple 9.5 est en forme SSA par construction, puisqu'on utilise des pseudo-registres frais à chaque instructions pour les traductions des expressions.

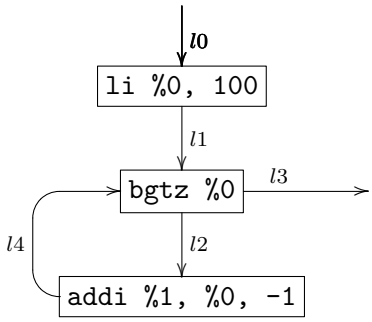
Exemple 9.9 *:* On veut traduire le programme suivant en forme SSA :

```
x := 100;
while x > 0 do
  x := x - 1;
```

La traduction naïve

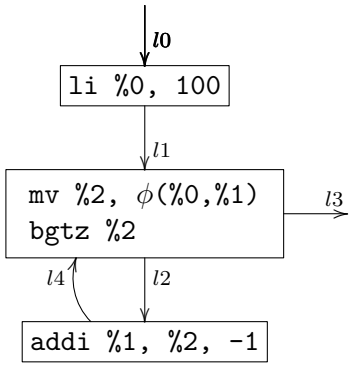


n'est pas en forme SSA car `%0` est affecté à deux endroits.



ne serait pas correct, car il faudrait utiliser `%1` au lieu de `%0` après le premier tour de

boucle. Par conséquent, on utilise une ϕ-fonction :



ϕ(%0,%1) correspondra à %0 si on vient de $l1$ et %1 si on vient de $l4$.

10. Explicitation des conventions d'appel

Les procédures et les fonctions constituent un des éléments de base des langages de haut niveau permettant la modularité du code (avec la possibilité de compilation séparée), l'abstraction de l'implémentation et la limitation de la portée des variables locales. De plus, le découpage en procédure peut également être utilisé par le système d'exploitation, par exemple pour gérer un système d'interruptions. Pour les langages de haut niveau, l'appel et le retour d'une fonction sont transparents. Pour les transformer en langage de bas niveau, il faut être capable de savoir comment gérer la pile d'appels, le passage des arguments, la valeur retournée, etc. Il existe plusieurs solutions : par exemple, l'ensemble des arguments peut-être écrit sur la pile d'appel, mais en général, pour des raisons d'efficacité, on utilise un petit nombre k de registres dédiés pour les k premiers arguments. Il est évident que l'appel d'une fonction doit respecter les même choix que ceux du corps de la fonction. Par conséquent, il est nécessaire de définir une convention d'appel entre

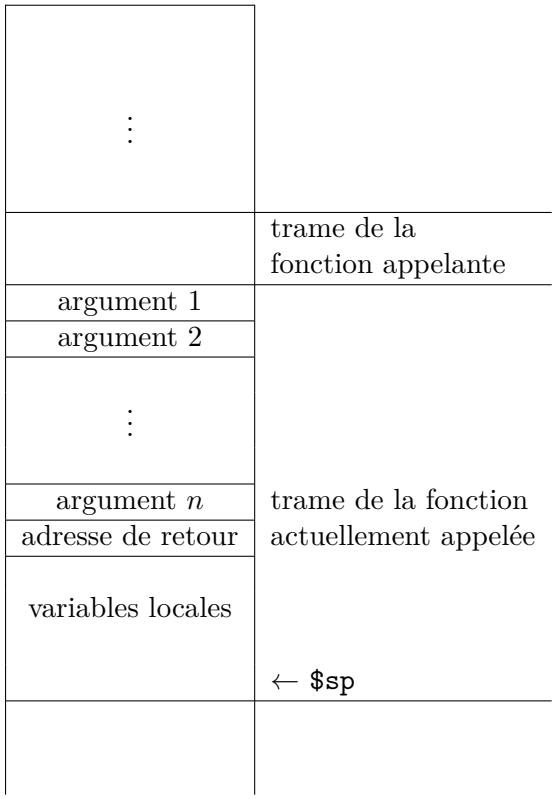
— la fonction appelante (*caller*) ;
— la fonction appelée (*callee*) ;
— le système d'exécution (*runtime system*) ;
— le système d'exploitation (*operating system*).

L'explicitation de la convention d'appel fait apparaître de nouvelles notions : registres physiques, trames de pile, emplacements de piles. Cela complique le langage intermédiaire, donc il vaut mieux le faire le plus tard possible. Néanmoins l'allocation de registres dépend de cette convention, donc il convient de l'expliciter immédiatement avant la phase d'allocation de registres (section 11).

10.1. Convention d'appel par pile

Cette convention d'appel est historiquement l'une des premières à avoir été utilisée, car c'est l'une des plus simple à mettre en œuvre. Dans cette convention, les paramètres des fonctions, l'adresse de retour, la valeur de retour et les variables locales d'une fonction sont stockées sur la pile d'appel.

Par convention, la pile d'appel grandit dans le sens des adresses décroissantes. Le sommet de la pile sera donc représenté en bas de celle-ci. À chaque appel correspond une partie de la pile appelée trame, ou bloc d'activation. Une trame de pile sera typiquement de la forme



Lors de l'appel d'une fonction, la fonction appelante met sur la pile les arguments de la fonction et l'adresse de retour (l'endroit où la fonction appelée doit retourner après l'appel, donc le code de l'instruction située juste après l'appel dans la fonction appelante). La fonction appelante passe ensuite le contrôle à la fonction appelée.

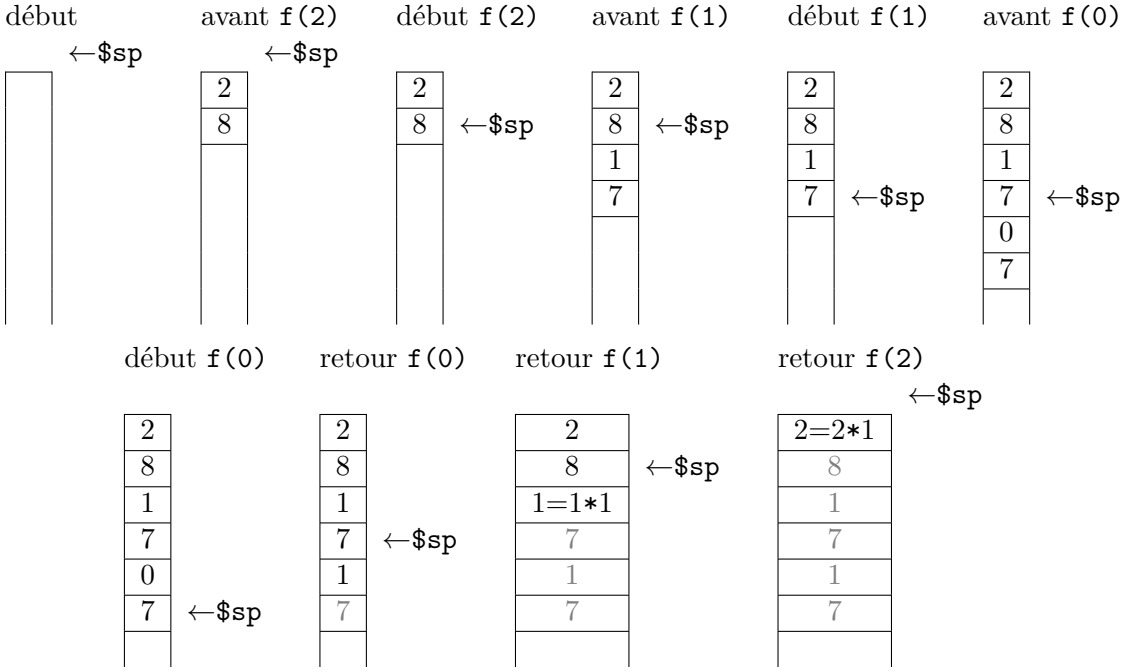
La fonction appelée crée sa trame (en décrémentant la valeur de `$sp` de la valeur adéquate). Elle exécute ensuite les instructions de son corps. Puis elle détruit sa trame (en réincrémentant la valeur de `$sp`), elle place la valeur de retour juste en dessous de la trame de l'appelante et elle saute à l'adresse de retour qui était indiquée dans sa trame.

Exemple 10.1 *:* On considère un programme Pseudo Pascal calculant la factorielle de 2 :

```
1 var r : integer
2 function fact(n : integer) : integer
3   if n <= 0
4   then
5     fact := 1
6   else
7     fact := fact(n-1) * n
8 r := fact(2)
```

Avec la convention d'appel par pile, la trame de `fact` contiendra 2 cases : une pour l'argument `n` et une pour l'adresse de retour.

Voici l'évolution de la pile au cours de l'exécution du programme (on note le numéro de ligne comme valeur pour l'adresse de retour).



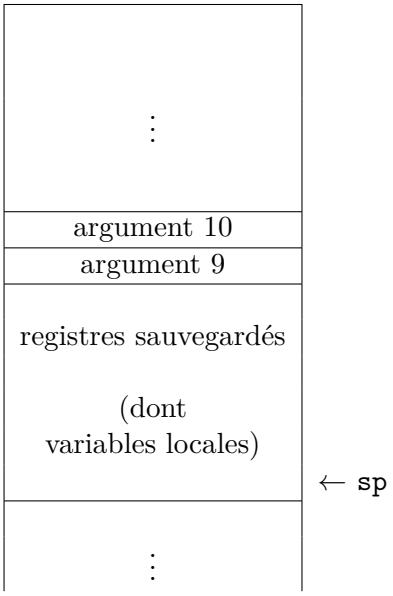
10.2. Convention d'appel de RISC-V

Avec la convention par pile, l'accès au variables et au paramètres des fonctions se fait sur la pile, donc en mémoire. Or l'accès à la mémoire est bien plus coûteux que l'utilisation d'un registre. Par conséquent, on a conçu d'autres conventions d'appel faisant appel à des registres, dont la convention d'appel définit dans le standard RISC-V.

Bien que les registres soient interchangeables (au niveau matériel), la convention d'appel des processeurs RISC-V leur donne un sens particulier. Le langage assembleur définit donc des noms plus facile à retenir (`ra sp a0`...) que leur numéro (néanmoins toujours accessible en utilisant le nom `x`*i*). On distingue plusieurs types de registres :

— les registres `a0` à `a7` servent à passer les huit premiers arguments lors de l'appel d'une fonction ; s'il y a plus de huit arguments, les suivants sont placés sur la pile ;
— les registres `a0` et `a1` servent également pour les valeurs de retour des fonctions ;
— les registres `t0` à `t6` peuvent être modifiés par l'appelé ; si l'appelant en a besoin, il doit les sauvegarder avant l'appel ; on les nomme donc *caller-saved* ;
— les registres `s0` à `s11` doivent être préservé par l'appelé ; si celui-ci les utilise, il doit sauvegarder leur valeur afin de les restaurer à la fin de l'appel ; on les nomme donc *callee-saved* ;
— le registre `sp` pointe sur le sommet de la pile (*stack pointer*) ;
— le registre `ra` contient l'adresse de retour, c'est-à-dire l'endroit du code où il faudra revenir après l'appel.

Dans cette convention la trame de pile sera typiquement de la forme

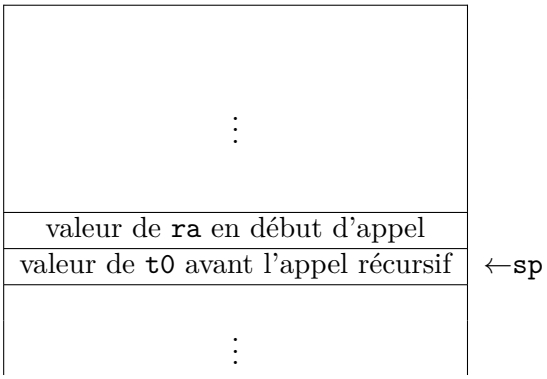


Les variables locales sauvegardées correspondront typiquement à des pseudo-registres qui auront été « spillés » lors de l'allocation.

Exemple 10.2 *:* La factorielle peut être codé avec le code suivant dans lequel les appels sont explicites :

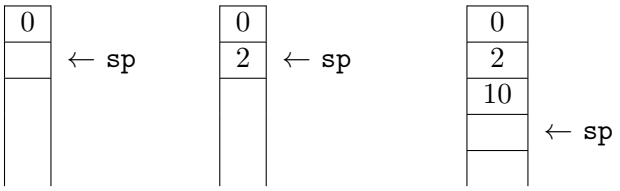
```
l1:  addi sp, sp, -8 -> l2
l2:  sw ra, 4(sp) -> l3
l3:  mv t0, a0 -> l4
l4:  bgtz t0 -> l7, l5

l5:  li a0, 1 -> l12

l7:  addi a0, t0, -1 -> l8
l8:  sw t0, 0(sp) -> l9
l9:  jal fact -> l10
l10: lw t0, 0(sp) -> l11
l11: mul a0, t0, a0 -> l12

l12: lw ra, 4(sp) -> l13
l13: addi sp, sp, 8 -> l14
l14: ret
```

Comme on le voit, les trames de la fonction `fact` sont de taille deux.

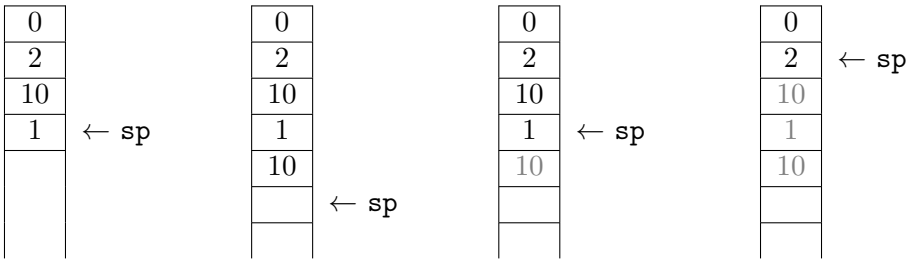


Voyons maintenant comment évolue la pile pendant l'appel à `fact 2`, en supposant que la pile est vide au départ.

`fact(2)`, point 4 `fact(2)`, point 9 `fact(1)`, point 4



`fact(1)`, point 9 `fact(0)`, point 4 `fact(1)`, point 12 `fact(2)`, point 12



On peut se demander ce que l'on gagne à utiliser les registres `a0` à `a7` pour passer les quatre premiers arguments plutôt que de les placer tous sur la pile (comme cela est fait dans la convention précédente). En effet, si la fonction appelle elle-même une autre fonction, il faut qu'elle sauvegarde auparavant les registres *caller-saved* `a0` à `a7`, et donc qu'elle les place sur la pile. Néanmoins :

— la fonction peut ne pas appeler de fonctions (on parle de procédure feuille) ;
— la fonction peut ne plus avoir besoin de ces registres après l'appel.

Dans ces deux cas il est inutile de les sauvegarder. Le gain est donc important surtout en présence d'instructions feuilles fréquemment invoquées.

Certaines compilateurs font une allocation de registres interprocédurale. Dans ce cas, il est possible de choisir une convention d'appel différente pour chaque procédure.

10.3. Explicitation des appels

Pour rendre explicite les appels de fonctions, il faut rajouter des instructions dans l'appelant pour préparer l'appel et en revenir, et des instructions au début et à la fin de l'appelé.

Concrètement, avant d'effectuer l'appel, l'appelant doit :
- mettre les quatre premiers arguments dans les registres `a0` à `a7`, et les suivants, s'il y en a, sur la pile ;
- enregistrer les registres *caller-saved* en les plaçant sur la pile : il s'agit des registres que l'appelé est autorisé à modifier, donc les registres `a0` à `a7`, `t0` à `t6` ; il n'y a bien entendu besoin de les sauvegarder que s'ils sont utilisés par l'appelant par la suite ;
- exécuter un instruction `jal` pour aller à la première instruction de l'appelé ; cette instruction met automatiquement l'adresse de la prochaine instruction de l'appelant dans le registre `ra`.

Au début de l'appelé, il faut alors :
- enregistrer les registres *callee-saved* en les plaçant sur la pile ; il s'agit des registres que l'appelé doit préserver, donc les registres `s0` à `s11` et de `ra` ; ici aussi, on ne les sauvegarde que s'ils sont redéfinis par l'appelé ;
- incrémenter `sp` de la taille de la trame pour qu'il pointe vers le sommet de la pile.

Remarque 10.1 *:* Le registre `ra` n'a besoin d'être sauvegardé que si l'appelé appelle lui-même une fonction.

À la fin de l'appelé il faut :
- placer la valeur de retour de la fonction dans `a0` ;
- restaurer la valeur de tous les registres *callee-saved* sauvegardés sur la pile.

Enfin, au retour de la fonction, dans l'appelant, il faut restaurer les registres *caller-save*.

Remarque 10.2 *:* À ce stade, l'allocation de registres n'a pas encore eut lieu. On ne peut donc a priori pas savoir si les pseudo-registres seront alloués dans des registres *caller-saved* ou *callee-saved*. Par conséquent on ne sait pas encore quelle sera la taille des trames. Une solution est de sauvegarder les registres physiques dans des pseudo-registres au lieu de sur la pile, et de se contenter de mettre des pseudo-instructions **`newframe`** en début d'appelé et **`delframe`** juste avant le retour. Ces pseudo-instructions seront transformées après l'allocation en code créant la trame et la détruisant. L'allocation de registre déterminera quels sont les pseudo-registres qui ont effectivement besoin d'être sauvegardés sur la pile, et lesquels peuvent être sauvegardés dans des registres réels.

La distinction entre registres *callee-saved* et *caller-saved* pourra être exploitée au moment de l'allocation de registres : on préfèrera allouer dans un registre *caller-saved* des pseudo-registres à courte durée de vie, par exemple des calculs intermédiaires qui ne seront pas utilisés à la suite de l'appel ; tandis que les pseudo-registres à longue durée de vie, typiquement ceux correspondant aux variables du programme de haut niveau, seront mieux utilisés dans des registres *callee-saved*.

Exemple 10.3 *:* Si on explicite les appels de l'exemple 9.1, on va obtenir le code suivant :

```
fact: newframe -> l1'  # on crée la trame de fact
l1':  mv %4, ra -> l2' # on sauvegarde ra
l2':  mv %0, a0 -> l6  # on récupère l'argument
l6:   li %1, 0  -> l5
l5:   blez %0  -> l4, l3
l3:   addi %3, %0, -1  -> l2
l2:   mv a0, %3 -> l3' # on prépare l'argument
l3':  jal fact -> l4'  # on effectue l'appel par un saut
l4':  mv %2, a0 -> l1  # on récupère la valeur retournée
l1:   mul %1, %0, %2  -> l0
l4:   li %1, 1  -> l0
l0:   mv a0, %1 -> l5' # on prépare la valeur de retour
l5':  mv ra, %4 -> l6' # on restaure la valeur de ra
l6':  delframe -> l7'  # on détruit la trame
l7':  ret              # on retourne à l'appelant
```

Remarque 10.3 *:* D'autres conventions d'appel, par exemple celle de gcc, utilisent un deuxième registre en plus de `sp`, appelé pointeur de trame (*frame pointer*), qui pointe sur la limite supérieure de la trame active. Ceci permet de pouvoir changer la taille de la pile au cours de l'exécution tout en ayant accès aux variables locales de façon statique.

10.4. Appels terminaux

Définition 10.1. *Un appel est à g dans un fonction h est dit* terminal *si cet appel est la dernière opération effectuée dans h.*

En particulier, le résultat de g devient celui de h.

Exemple 10.4 *:* Dans

```
function fact (n : integer) : integer;
begin
  if n <= 0 then
    fact := 1
  else
    fact := n * fact(n-1)
end
```

l'appel récursif à `fact` n'est pas terminal. Pour obtenir une version avec un appel terminal, il faut utiliser un accumulateur :

```
function fact (n, accu : integer) : integer;
```

```
begin
  if n <= 0 then
    fact := accu
  else
    fact := fact(n-1, n * accu)
end
```

Il est possible d'optimiser les appels terminaux. En effet, après le retour d'un appel terminal, l'appelant se contentera de détruire sa trame puis de passer la main à son propre appelant. Par conséquent, la trame de l'appelant n'a plus lieu d'être avant même l'appel, et il vaut mieux que l'appelé rende directement la main à l'appelant de l'appelant.

Supposons que l'on ait un appel terminal à g dans h. Celui-ci sera compilé de la manière optimisée suivante :

— la valeur initiale des registres *callee-saved* est restaurée (y compris `ra`, qui contient donc l'adresse de retour dans l'appelant de h) ;
— la trame de h est désallouée ;
— les arguments de g sont passés comme d'habitude, les quatre premiers dans les registres `a0` à `a7`, les autres sur la pile ;
— le contrôle est tranféré à g par un simple saut (`j` au lieu de `jal`).

Du point de vue de l'appelé g, tout se passe comme s'il était appelé par l'appelant de h.

Exemple 10.5 *:* À la fin de la version terminale de l'exemple 10.4, on aura le code suivant :

```
mul a1, a0, a1      # calcul des arguments
addi a0, a0, -1
lw ra, 0(sp)        # restauration des callee-saved
addi sp, 4          # désallocation de la pile
j fact              # appel de fact
```

Dans le cas d'un appel récursif terminal, il est possible d'optimiser encore plus. En effet, si f s'appelle elle-même de façon terminale, rien ne sert de restaurer les registres *callee-saved*, détruire la trame de f pour ensuite recréer la trame et sauvegarder les registres *callee-saved* dans l'appelé. Il suffit de passer le contrôle à f directement après l'allocation de la trame dans f. Il convient donc de :

— placer les arguments attendus par f, dans les registres `a0` à `a7` pour les premiers, et en écrasant les précédents dans la trame de l'appelant pour les autres ;
— transférer le contrôle par un simple saut `j` au point situé après l'allocation de la trame et la sauvegarde des registres *callee-saved*.

Exemple 10.6 *:* Le début de la compilation de la version terminale de l'exemple 10.4 sera

```
fact: addi sp, sp, -4 -> l2  # création de la trame
l2:   sw ra, 0(sp)    -> l3  # sauvegarde des callee-saved
l3:   blez a0         -> l4, l5
```

La fin deviendra

```
l10:  mul a1, a0, a1  -> l11 # calcul des arguments
l11:  addi a0, a0, -1 -> l3  # appel de fact : retour à l3
```

Ce code correspond exactement à celui qui serait produit en utilisant une boucle

```
function fact (n, accu : integer) : integer;
begin
  while not (n <= 0) do
  begin
    accu := n * accu;
    n := n - 1
  end;
  fact := accu
end
```

On comprend à partir de cet exemple combien l'optimisation des appels terminaux joue un rôle central pour la compilation des langages fonctionnels, car elle permet d'obtenir les mêmes performances qu'une version impérative. On comprend aussi l'intérêt d'écrire et d'utiliser des fonctions dont les appels récursifs sont terminaux (cf. par exemple la documentation de la librairie standard `List` en OCaml).

L'optimisation des appels terminaux existe, avec certaines restrictions, sur la plate-forme .net de Microsoft. Elle n'existe pas en C ou en java, mais certains chercheurs se sont évertués à trouver des moyens de la simuler Schinz and Odersky [2001].

10.5. Fonctions imbriquées

Dans de nombreux langages (mais pas en Pseudo-Pascal), il est possible de définir des procédures locales. Par exemple :

```
function fact(n : integer) : integer;
var accu : integer
  function aux(n : integer) : integer;
  begin
     if n <= 0 then
       aux := accu
     else
     begin
       accu := n * accu;
       aux := aux(n-1)
     end;
  end;
begin
  accu := 1;
```

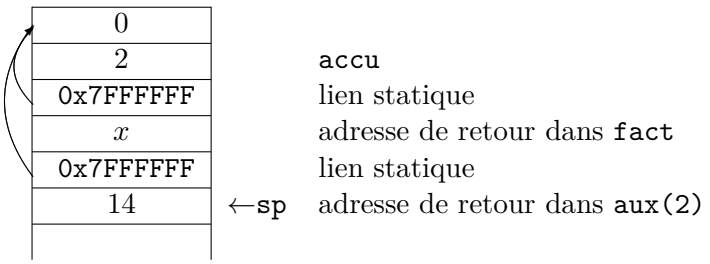
```
  fact := aux(n)
end;
```

Comme on le voit, la fonction `aux` peut accéder aux variables locales (ici, `accu`) de la fonction dans laquelle elle est imbriquée. Pour cela, il faut que la fonction imbriquée ait accès à la trame de la fonction englobante. On utilise par conséquent un paramètre supplémentaire appelé lien statique qui pointe vers la trame de la fonction englobante. Quand `aux` est appelé, il faut lui passer ce lien.

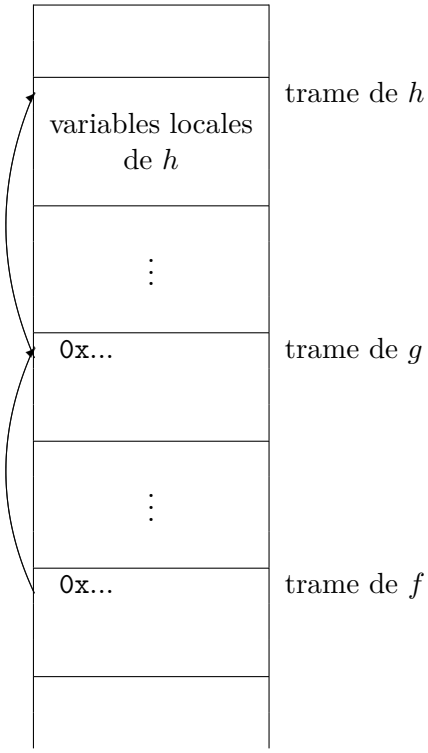
Exemple 10.7 *:* En compilant `aux` ci-dessus (sans optimisation des appels terminaux), on obtient le code suivant :

```
    addi sp, sp, -8
    sw ra, 0(sp)
    bgtz a0 -> 7,4
4:  lw t0, 4(sp)        # lien statique
    lw v0, -4(t0) -> 14 # valeur de accu dans la trame de fact


7:  lw t0, 4(sp)        # lien statique
    lw v0, -4(t0)       # valeur de accu dans la trame de fact
    mul v0, a0, v0
    sw v0, -4(t0)
    addi a0, a0, -1
    sw t0, -4(sp)       # préparation de la trame de l'appelé
    jal fact -> 14
14: lw ra, 0(sp)
    addi sp, sp, 8
    jr ra
```

Lors de l'appel à `fact(2)`, après l'allocation de la trame de `aux(1)`, la pile sera :



On le voit, le lien statique peut remonter arbitrairement dans la pile, en particulier en cas d'appel récursif. De plus, si on considère une fonction f imbriquée dans une fonction g imbriquée dans une fonction h, f peut accèder aux variables locales de h en suivant son lien statique qui l'amène à la trame de g, dans lequel on peut lire le lien statique de g qui pointe sur la trame de h.



Les liens statiques forment donc en général une liste chaînée.

On peut également autoriser de passer des fonctions comme arguments, par exemple :

```
procedure iterate (t : array of integer; n : integer;
                   procedure f (i : integer) );
var i : integer;
begin
  for i := 1 to n do
    f(t[i])
end

function sum (t : array of integer; n : integer) : integer;
var s : integer;
  procedure add (x : integer);
  begin
    s := s + x
  end;
begin
  s := 0;
  iterate (t, n, add);
  sum := s
end;
```

Dans ce cas, quand `sum` passe `add` comme argument à `iterate`, il lui passe l'adresse du code de `add` mais également l'adresse de sa propre trame de pile qui sera utilisée comme lien statique lors des appels `f(t[i])`. `iterate` attend donc à la fois un pointeur vers le code de `f` mais également le lien statique correspondant à `f`.

Dans les langages fonctionnels, les fonctions sont des objets de première classe et peuvent donc être retournées par une fonction ou stockées dans des variables, etc. La façon standard de compiler de tels objets est de fournir un couple constitué d'un pointeur vers le code de la fonction, et de son environnement. Cf. le tutoriel de Xavier Leroy *An introduction to compiling functional languages* pour plus de détails sur la compilation de langages fonctionnels (http://pauillac.inria.fr/~xleroy/talks/tutorial-tic98.ps.gz).

11. Allocation de registres

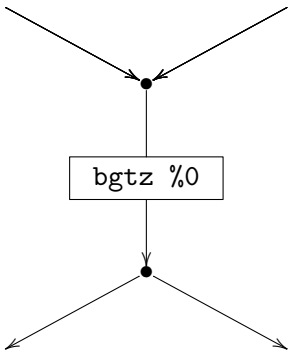
On a jusqu'à présent travaillé avec des pseudo-registres, qui sont en nombre potentiellement infini, alors que les registres physiques sont eux en nombre fini. L'allocation de registres consiste à dire quel registre physique sera utilisé à la place d'un pseudo-registre. Il arrive fréquemment que le nombre de registres physiques sont insuffisant pour que tous les pseudo-registres soient alloués. Dans ce cas, il faut utiliser la mémoire pour stocker la valeur, typiquement sur la pile d'appel, et on dit que le pseudo-registre est « spillé ».

11.1. Analyse de la durée de vie

Il est évident que le nombre de registres physique est bien inférieur à celui des pseudo-registres d'un programme ordinaire. Néanmoins, les pseudo-registres ne sont parfois plus utilisés à partir d'un certain point, et il est donc possible d'allouer le même registre à deux pseudo-registres qui ont des durée de vie différente.

Une variable représentera soit un pseudo-registre, soit un registre physique qui n'est pas réservé à un usage spécial (comme `sp`).

Définition 11.1. *À chaque nœud du graphe de flot de contrôle on associe deux points de programme, l'un à l'entrée du nœud, l'autre à la sortie.*



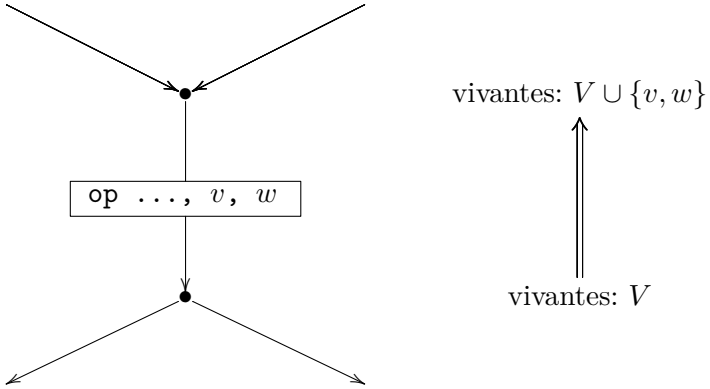
Une variable v est dite vivante *au point p s'il existe un chemin menant de p à un point p' où v est utilisée, et si v n'est pas définie le long de ce chemin.*
Sinon, la variable est dite morte.

L'analyse de durée de vie consiste à déterminer à quels points du programme une variable est vivante, de manière à savoir quels variables interfèrent et doivent donc être allouées à des registres différents. L'analyse de durée de vie est une approximation : si

par exemple une variable est utilisée dans une branche qui ne sera jamais atteinte (par exemple, si une condition d'un `if` n'est jamais vérifiée), elle sera quand même déclarée comme vivante. De ce fait, « vivante » signifie « potentiellement vivante » tandis que « morte » signifie « certainement morte ». Par conséquent, l'approximation est sûre : au pire, on allouera à deux variables des registres distincts alors qu'elles auraient pu partager le même.

Pour calculer les variables vivantes en un point, on va considérer celles qui sont vivantes au point suivant.

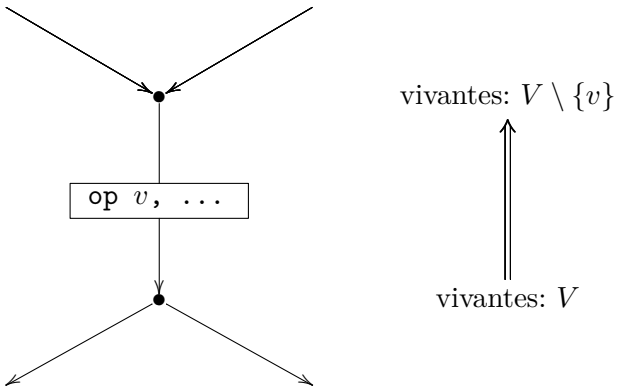
— Une variable v est engendrée par une instruction i si i utilise v, c'est-à-dire i lit une valeur de v. Dans ce cas, v est vivante dans le point qui précède immédiatement i.



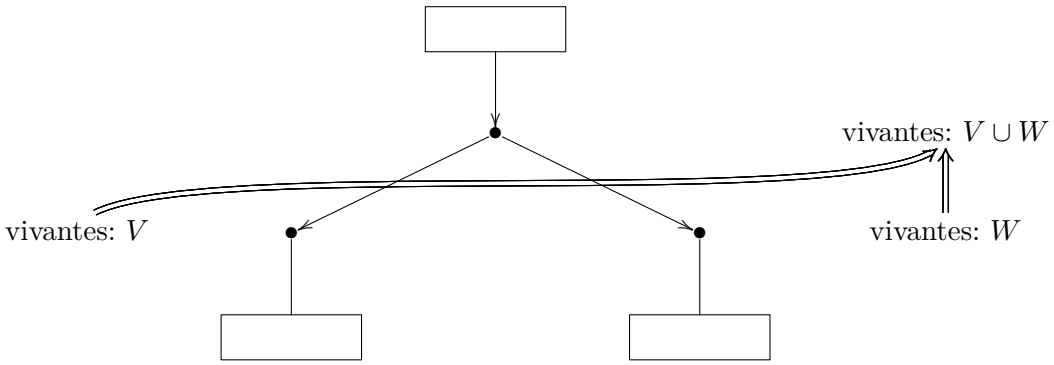
— Les registres `a0` à `a7` sont engendrés par un appel de fonction. (La fonction appelée est susceptible d'utiliser ses arguments.)

— Les registres `ra` et `a0`, ainsi que tous les registres *callee-saved* sont engendrés par un retour de fonction. (L'adresse de retour est utilisée pour y retourner, et on suppose que la valeur retournée est utilisée par l'appelant.)

— Une variable v est tuée par une instruction i si i définit v, c'est-à-dire i place une valeur de v. Dans ce cas, v est morte dans le point qui précède immédiatement i (sauf si i engendre v).



— Les registres *caller-saved* sont tués par un appel de fonction.
— Si i n'engendre ni ne tue v, alors v est vivante au point précédant i ssi elle est vivante au point suivant.
— Une variable est vivante après i ssi elle est vivante avant l'un des successeurs de i.



Les ensembles $\text{Vivantes}_{\text{avant}}(i)$ des variables vivantes avant l'instruction i et $\text{Vivantes}_{\text{après}}(i)$ des variables vivantes après l'instruction i vérifient donc les équations suivantes :

$$\begin{aligned} \text{Vivantes}_{\text{avant}}(j) &\subseteq \text{Vivantes}_{\text{après}}(i) \qquad \text{si } i \to j \\ (\text{Vivantes}_{\text{après}}(i) \setminus \text{Tuées}(i)) \cup \text{Engendrées}(i) &\subseteq \text{Vivantes}_{\text{avant}}(i) \end{aligned}$$

Toute solution de ces inéquations est sûre. La plus petite est celle qui donnera les meilleurs résultats, puisque moins de variables seront déclarées vivantes, moins d'interférence il y aura.

La recherche de solution conduit à une analyse en arrière : la vivacité se propage dans le sens inverse des arêtes du graphe de flot de contrôle.

On peut voir ces inéquations comme une inéquation de la forme $F(x) \sqsubseteq x$ où F est la fonction qui va de l'ensemble des fonctions de l'ensemble des points du programme dans l'ensemble des parties de l'ensemble des variables vers lui-même, avec :

$$\begin{aligned} F(f)(i_{\text{après}}) &= \bigcup_{i \to j} f(j_{\text{avant}}) \\ F(f)(i_{\text{avant}}) &= (f(i_{\text{après}}) \setminus \text{Tuées}(i)) \cup \text{Engendrées}(i) \end{aligned}$$

On peut vérifier que cette fonction est bien monotone, et on peut donc appliquer les résultats de la section A.2 pour calculer le point fixe : on assigne à chaque point un ensemble vide de variables, puis on applique F en remontant le graphe de flot de contrôle, jusqu'à arriver à un point fixe.

Pour l'analyse de la durée de vie, la propriété à un point p ne change que si celle d'un de ses successeurs change. Par conséquent on peut utiliser un algorithme moins naïf :

```
W ← { points de contrôle }
tant que W ≠ ∅
```

```
p ← pop W
f(p) ← F(f)(p)
si f(p) ⊃ \old{f(p)}
  push W (pred(p))
```

Exemple 11.1 *:* Nous allons effectuer l'analyse de durée de vie sur le code de l'exemple 10.3.

Pour arriver plus vite au point fixe, on peut effectuer l'analyse de durée de vie en choisissant le point `p` en remontant le long du graphe de flot de contrôle. On obtient le résultat suivant :

i	Vivantes$_{\text{après}}(i)$	Vivantes$_{\text{avant}}(i)$
l7'		ra a0 s0–s11
l6'	ra a0 s0–s11	ra a0 s0–s11
l5'	ra a0 s0–s11	%4 a0 s0–s11
l0	%4 a0 s0–s11	%1 %4 s0–s11
l4	%1 %4 s0–s11	%4 s0–s11
l1	%1 %4 s0–s11	%0 %2 %4 s0–s11
l4'	%0 %2 %4 s0–s11	%0 %4 a0 s0–s11
l3'	%0 %4 a0 s0–s11	%0 %4 a0 s0–s11
l2	%0 %4 a0 s0–s11	%0 %3 %4 s0–s11
l3	%0 %3 %4 s0–s11	%0 %4 s0–s11
l5	%0 %4 s0–s11	%0 %4 s0–s11
l6	%0 %4 s0–s11	%0 %4 s0–s11
l2'	%0 %4 s0–s11	%4 a0 s0–s11
l1'	%4 a0 s0–s11	ra a0 s0–s11
fact	ra a0 s0–s11	ra a0 s0–s11

L'analyse de la durée de vie fait partie de la famille des analyses de flot de données. Ces analyses associent une propriété à chaque points du code. En général, les propriétés sont ordonnés par un treillis. Un système d'inéquations définit un ensemble de solutions sûres, dont on cherche la plus petite. Quelques exemples :

— quelles variables ont une valeur connue ?
— quelles sont les relations affines connues entre variables ?
— quelles variables sont certainement égales ?
— quelles expressions seront certainement évaluées ?
— quels points du code ont certainement été atteint auparavant ?

11.1.1. Élimination du code mort

On dit qu'une instruction est pure si elle n'a pas d'autre effet que de modifier sa variable de destination. Par exemple, `li` ou `add` sont pures, mais `div` ou `jal` ne le sont pas. (`div %0, %1` place le quotient et le reste de la division de `%0` par `%1` dans deux registres `%lo` et `%hi`.)

Si la variable de destination d'une instruction pure est morte à la sortie de l'instruction, cela veut dire qu'elle n'est pas utilisée par la suite. On peut donc éliminer l'instruction sans modifier la sémantique du programme. Plus généralement, on peut supprimer une instruction si toutes les variables tuées par l'instruction sont mortes après l'instruction.

Il semble peu probable qu'un programmeur écrive de lui-même un programme dans lequel une variable est affectée mais pas utilisée. Néanmoins, il y a plusieurs cas dans lesquels de telles instructions ont pu être générées par le compilateur : on peut avoir des instructions qui correspondent aux initialisations des variables, comme l'instruction `16` de l'exemple 10.3. On voit que la variable `%1` est morte après l'instruction, donc cette instruction est du code mort et peut être éliminée.

De telles instructions éliminables peuvent aussi avoir été produites lors de la suppression des calculs redondants (section 9.3).

Exemple 11.2 *:* Si on reprend l'exemple 9.7, l'analyse de durée de vie donne :

Instr.	Variables vivantes
	%2 %3
1	%2 %4
2	%4 %5
3	%1 %4 %5
4	%1 %4 %5
5	%1 %4 %5 %7
6	%1 %4 %5 %7
7	%1 %4 %5 %7 %9
8	%1 %4 %5 %7 %10
9	%1 %4 %5
10	%1 %5
11	%1 %12
12	

On voit que si à la sortie de l'instruction 4 (resp. 6 et 10), la variable `%6` (resp. `%8` et `%11`) sont mortes. On peut donc supprimer ces instructions.

La suppression des instructions mortes est susceptible de modifier la durée de vie des variables si elles étaient utilisées dans les instructions éliminées. Il faut donc relancer l'analyse de durée de vie, et de nouvelles instructions peuvent être devenue mortes.

11.2. Graphe d'interférence

Définition 11.2 (Interférence)**.** *Deux variables* interfèrent *si l'une est vivante à la sortie d'une instruction définissant l'autre.*

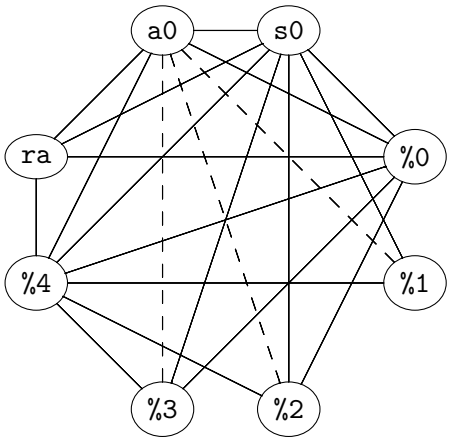
Le graphe d'interférence est un graphe dont les nœuds sont les variables et qui ont une arête entre les variables qui interfèrent.

Deux variables qui n'interfèrent pas peuvent être placées dans le même registre, tandis que deux variables qui interfèrent doivent obligatoirement être placées dans des registres distincts.

Il y a cependant une exception. Quand une variable x est vivante à la sortie d'une instruction définissant une autre y et que la valeur reçue par y est obligatoirement celle de x, il n'y a pas lieu de considérer que les variables interfèrent. Au contraire, on aimerait alors utiliser le même registre pour les deux variables. Cette propriété est en général indécidable, mais il existe un cas particulier simple, celui d'une instruction `mv y, x`.

Par conséquent, on ajoute dans le graphe d'interférence des arêtes de préférence indiquant qu'on souhaite allouer le même registre à deux nœuds.

Exemple 11.3 *:* Dans le cas du programme des exemples 10.3 et 11.1, en supposant qu'on a éliminé l'instruction morte `l6`, on obtient les interférences suivantes (les pointillés représentent les arêtes de préférence) :



11.2.1. Coloriage de graphe

Supposons que l'on dispose de k registres physiques allouables, et du graphe d'interférence d'un programme. Le problème de l'allocation de registres semble se résumer à :

— attribuer une couleur parmi k à chaque sommet représentant un pseudo-registre
— de façon à ce que deux sommets reliés par une arête d'interférence ne reçoivent jamais la même couleur
— et si possible de façon à ce que deux sommets reliés par une arête de préférence reçoivent la même couleur.

On cherche donc à résoudre le problème du k-coloriage de graphe.

Définition 11.3 (k-coloriage de graphe)**.** *k étant fixé, le problème du k-coloriage prend comme entrée un graphe (E, V) et décide s'il est possible d'affecter un entier $1 \leq c(x) \leq k$ à chaque sommet $x \in E$ de façon à ce que pour tout $x, y \in V$ on ait $c(x) \neq c(y)$.*

Néanmoins, quelques problèmes demeurent :

— pour $k \geq 3$, le problème du k-coloriage de graphe est NP-complet ;

— si le graphe n'est pas k-coloriage, il faut savoir quelles variables seront affectées à un registre et quelles variables seront spillées, c'est-à-dire placées sur la pile ;
— assez souvent, les registres des machines ne sont pas tous indépendants et interchangeables — on peut par exemple avoir des registres différents pour les entiers et les flottants.

L'algorithme le plus simple pour colorier un graphe a été proposé par Chaitin en 1981. Il part de la constatation suivante : un sommet s de degré strictement inférieur à k est *trivialement colorable*, dans le sens où G est k-colorable si et seulement si $G \setminus \{s\}$ est k-colorable. L'idée est de donc de supprimer de façon itérative tous les sommets de degrés strictement inférieurs à k, et de spiller un sommet quand ce n'est pas possible.

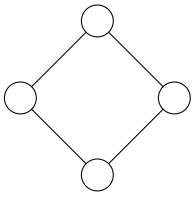
```
procédure Colorier(G)
  si il existe un sommet s trivialement colorable
  alors
    Colorier(G\{s})
    attribuer une couleur disponible à s
  sinon si il existe un sommet s
  alors
    Colorier(G\{s})
    spiller s
  sinon
    renvoyer le coloriage vide
```

Le choix d'un sommet à spiller est critique :
— pour une meilleure efficacité, il faut choisir un pseudo-registre peu utilisé (l'« utilisation » d'un pseudo-registre peut être déterminée à l'aide d'une analyse de flot de données) ;
— pour faciliter la suite du coloriage, il vaut mieux choisir un sommet avec un grand degré.

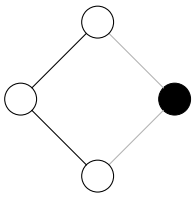
Pour choisir quel sommet spiller, on attribue un coût à chacun en fonction de ces critères, et on choisit celui de plus bas coût.

Néanmoins, l'algorithme de Chaitin est souvent pessimiste :

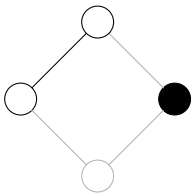
Exemple 11.4 *:* On veut 2-colorier le graphe suivant :



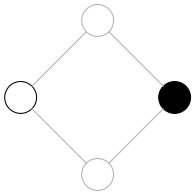
Aucun des nœuds n'est trivialement colorable. Par conséquent, on n'en élimine un que l'on décide de spiller.



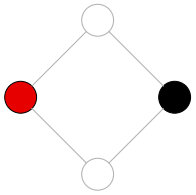
Ici, on a deux nœuds trivialement colorables. On en choisit un et on essaie de colorier le graphe résultant :



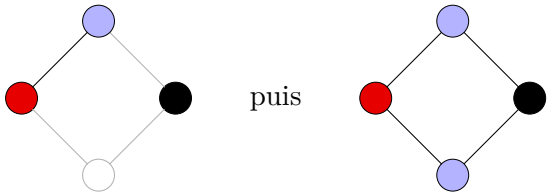
On recommence :



On recommence, on obtient le graphe vide avec le coloriage vide. On peut associer le nœud restant avec une des deux couleurs :



Ensuite, on n'a plus le choix pour colorier les nœuds suivants :



Comme on le voit, le dernier nœud aurait pu ne pas être spillé.

Il est possible d'améliorer l'algorithme de Chaitin pour ne spiller les nœuds que si nécessaire.

```
procédure Colorier(G)
  si il existe un sommet s trivialement colorable
```

```
  alors
    Colorier(G\{s})
    attribuer une couleur disponible à s
  sinon si il existe un sommet s
  alors
    Colorier(G\{s})
    si il existe une couleur disponible pour s
    alors
      attribuer cette couleur à s
    sinon
      spiller s
  sinon
    renvoyer le coloriage vide
```

On aimerait également prendre en compte les arêtes de préférence dans le coloriage. Une technique pour cela est de fusionner (*coalesce*) les nœuds reliés par une arête de préférence. Si cette fusion fait se recouvrir une arête d'interférence et de préférence, alors la première est prioritaire. Néanmoins, il peut être problématique de fusionner tous les nœuds possible, car cela peut créer des nœuds de très grand degré, qu'il ne sera donc pas possible de k-colorier alors que cela était possible avant la fusion. Par conséquent, on définit des critères qui garantissent que la fusion préserve la k-colorabilité du graphe.

Critère de Briggs Deux sommets peuvent être fusionnés si le sommet résultant à moins de k voisins non trivialement colorable.

Critère de George Deux sommets peuvent être fusionnés si tout voisin non trivialement coloriable de l'un est également voisin de l'autre.

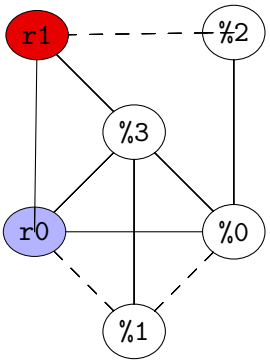
Fusionner deux nœuds peut rendre un nœud trivialement colorable. Réciproquement, simplifier un graphe en retirant un nœud trivialement colorable peut rendre deux nœuds fusionnables. Il faut donc alterner les deux phases. Néanmoins, on va chercher à éviter de simplifier un nœud qui pourrait être fusionné ensuite (c'est-à-dire un nœuds d'où part une arête de préférence). Si jamais cela n'est pas possible, et qu'aucune fusion n'est acceptable non plus, alors on « gèle » une arête de préférence (c'est-à-dire qu'on la supprime) pour éventuellement pouvoir simplifier des nœuds.

George et Appel proposent donc l'algorithme suivant :

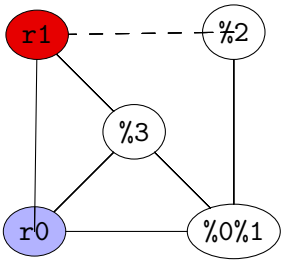
```
procédure Colorier(G)
  Simplifier(G)

procédure Simplifier(G)
  si il existe un sommet s trivialement colorable
     et aucune arête de préférence ne sort de s
  alors
    Simplifier(G\{s})
    attribuer une couleur disponible à s
  sinon
```

```
    Fusionner(G)

procédure Fusionner(G)
  si il existe une arête de préférence a—b
    et a—b respecte le critère pour la fusion
  alors
    G' ← G où un unique sommet ab remplace a et b
    Simplifier(G')
    attribuer à a et b la couleur attribuée à ab
  sinon
    Geler(G)

procédure Geler(G)
  si il existe un sommet s trivialement colorable
  alors
    G' ← G privé des arêtes de préférence issues de s
    Simplifier(G')
  sinon
    Spiller(G)

procédure Spiller(G)
  si il existe un sommet s de G
    et le coût de s est minimal
  alors
    Simplifier(G\{s})
    si il existe une couleur disponible pour s
    alors
      attribuer cette couleur à s
    sinon
      spiller s
  sinon
    renvoyer le coloriage vide
```

Exemple 11.5 *:* On part du graphe d'interférence suivant, que l'on veut 2-colorier. (Deux nœuds sont déjà coloriés, car ce sont les variables correspondant aux registres physiques.)

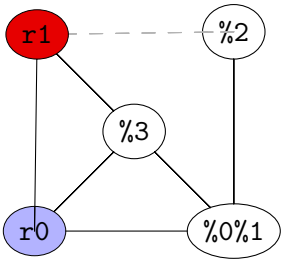


Il est impossible de simplifier un nœud à cause des arêtes de préférence. On peut fusionner `%0` et `%1` : en effet, les voisins non trivialement colorables de `%1`, soit `%3`, sont tous des voisins de `%0` (critère de George). On obtient le graphe :

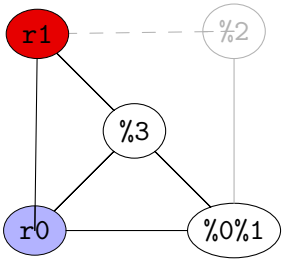


On peut remarquer que l'arête de préférence `%1`−`r0` disparaît au profit de l'arête d'interférence `%0`−`r0` lors de la fusion.

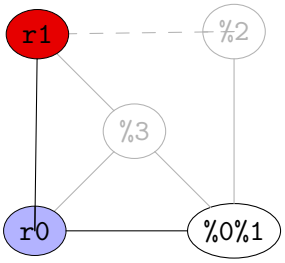
Aucun nœud ne peut être simplifié. Les critères de Briggs et de George ne s'appliquent pas non plus. Par conséquent, il faut geler les arêtes de préférence partant de `%2` pour obtenir :



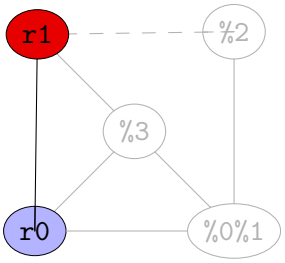
On peut maintenant simplifier `%2` :



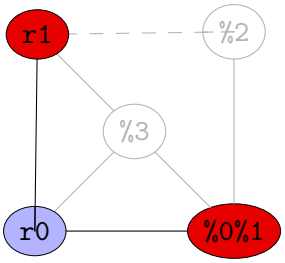
Aucune simplification ni fusion n'est possible, on décide de spiller `%3` (plus grand degré possible) :



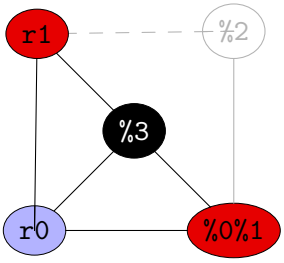
On peut simplifier `%0%1` :

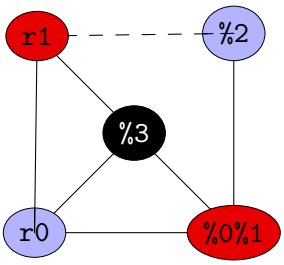


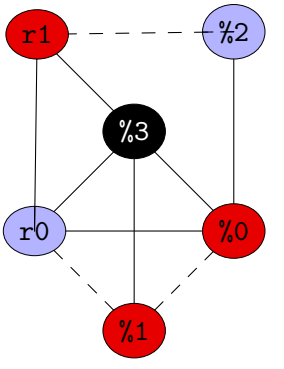
`r0` et `r1` sont déjà coloré, on reprend la série de transformations qui nous a menés jusqu'ici, dans l'ordre inverse, pour attribuer les couleurs :



On voit qu'on est vraiment obligé de spiller `%3` :

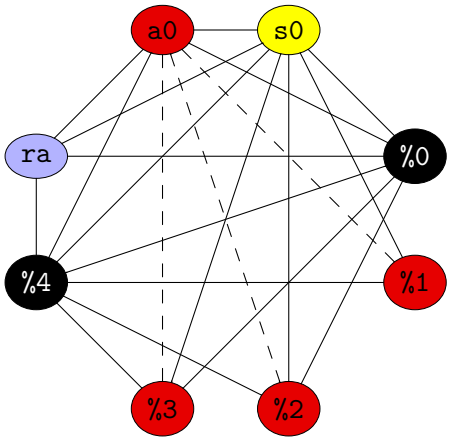






Exercice 11.1 *:* 3-colorier le graphe de l'exemple 11.3. Attention, les couleurs de `ra`, `a0`

et `s0` sont déjà attribuées. Une solution possible est :



On voit qu'on a dû spiller `%0` et `%4`. On les stocke donc en mémoire aux adresses `0(sp)` et `4(sp)`. La trame de `fact` est donc de taille 8.

Les arêtes de préférence ont pu être respectées et `%1`, `%2` et `%3` seront stockés dans `a0`. Par conséquent, les instructions `l2`, `l4'` et `l0` deviennent inutiles et peuvent être supprimées.

11.2.2. Spill

Si certains sommets n'ont pu être colorés, il faut modifier le code pour diminuer le nombre de registres physiques nécessaires simultanément. La solution la plus simple consiste, pour chaque pseudo-registre spillé, à réserver un emplacement de la trame de pile où sera stockée la valeur du pseudo-registre.

Le problème est que pour des machines cibles de type RISC comme le RISC-V, seules des instructions dédiées peuvent accéder à la mémoire (`lw` et `sw` pour le RISC-V). Il faut donc utiliser ces instructions pour accéder à la valeur des variables spillés. Or ces instructions ont besoin d'utiliser des registres physiques !

Exemple 11.6 *:* Si `%0` et `%2` sont spillés, on voudrait remplacer `add %0, %1, %2` par

```
lw %2, 4($sp)
add %0, %1, %2
sw %0, 0($sp)
```

Le problème est qu'on ne sait pas quels registres physiques utiliser pour `%0` et `%2`.

Heureusement le fait de rajouter les instructions pour stocker les valeurs sur la pile permet de changer la durée de vie des variables spillées. Il est alors possible de relancer l'allocation de registre, en interdisant de spiller les pseudo-registres ajoutés pour les instructions `lw` et `sw` afin de garantir la terminaison.

Il existe une autre solution, plus simple, qui consiste à réserver pour charger et sauvegarder les valeurs de pseudo-registres spillés deux registres que l'on rend non allouables.

Deux suffisent, car il y a au plus deux registres sources et un registre destination dans une instruction. Cette solution n'est viable que quand on dispose d'un assez grand nombre de registres physiques, ce qui est justement un des avantages d'une architecture RISC.

Il est bien entendu possible d'apporter des améliorations à cette solution simple pour minimiser le nombre de chargement et de sauvegarde des variables.

Exemple 11.7 *:* En appliquant le coloriage obtenu dans l'exemple 11.1 sur le code de l'exemple 10.3, on obtient donc le code :

```
fact: addi sp, sp, -8 -> l1' # création de la trame
l1':  sw ra, 4(sp) -> l2'    # on sauvegarde ra
l2':  sw a0, 0(sp) -> l5     # on sauvegarde a0
l5:   blez a0 -> l4, l3
l3:   addi a0, a0, -1 -> l3'
l3':  jal fact -> l8'        # on effectue l'appel par un saut
l8':  lw t0, 0(sp) -> l1     # on restaure la valeur de a0
l1:   mul a0, t0, a0   -> l0
l4:   li a0, 1  -> l0
l0:   lw ra, 4(sp) -> l6'    # on restaure la valeur de ra
l6':  addi sp, sp, 8 -> l7'  # on détruit la trame
l7':  ret                    # on retourne à l'appelant
```

12. Compléments

Dans ce cours, nous avons vu différents aspects d'un compilateur. Néanmoins, il reste de nombreux domaines que nous aurions pu étudier également :

Analyse sémantique il s'agit de vérifier de façon statique que le programme source est correct d'un point de vue sémantique. Un exemple simple est la vérification de type, mais on peut également envisager de vérifier que des préconditions sont bien validées, qu'il n'y a pas d'accès en dehors des bornes d'un tableau, etc.

Types de données Dans ce cours, on n'a considéré qu'un seul type de données, les entiers, mais en pratique on a plusieurs tailles possibles, des flottants, des conversions implicites et explicites, ...Il est souvent également possible pour le programmeur de définir ses propres types.

Linéarisation du code Il s'agit de passer du graphe de flot de contrôle à un code linéaire, dans lequel les instructions se succèdent sauf en cas de branchement explicite. Un des objectif est de minimiser le nombre de sauts inconditionnels. Il faut également remplacer les instructions `newframe` et `endframe` par celle créant et détruisant la trame de la pile.

Gestion de la mémoire Il faut tout d'abord expliciter comment allouer des objets dans le tas (`new array` en pseudo-pascal, `malloc` en C, `new` en java). On peut alors choisir si ces objets doivent être désalloués manuellement (`free` en C) ou automatiquement. Dans le premier cas, le programmeur risque de faire des erreurs : désallouer un objet qui sera utilisé par la suite ou désallouer deux fois un objet. Ces erreurs ne sont souvent pas détectées immédiatement et conduisent à des comportements anormaux difficiles à analyser. Par conséquent, de nombreux langages modernes utilisent un ramasseur de miettes (ou glaneur de cellules, *garbage collector*) dont le but est de désallouer automatiquement les objets du tas qui ne sont plus accessibles. Un tel ramasseur de miette interagit avec le code produit par le compilateur.

Exceptions Ajouter un mécanisme d'exceptions dans le langage source va rendre possible des remontées arbitrairement dans la pile d'appel, jusqu'au gestionnaire d'exceptions le plus proche. On aimerait que ceci puisse se faire en temps constant.

Objets Dans un langage à objets, les portées des noms ne sont pas définies par rapport à la hiérarchie du code, mais par rapport aux données. Chaque objet comportera un pointeur vers sa classe qui comportera des pointeurs vers les variables de classes (variables *static* en java) et les méthodes de la classe, ainsi que vers la (les) classe(s) dont elle hérite. L'accès à une variable dans une méthode doit donc éventuellement remonter le long de ces pointeurs.

Optimisations Il existe bien d'autres optimisations dont nous n'avons pas parlé dans ce cours : fusion de boucles, *inlining* de fonctions, dépliage de boucles, ...

La compilation est un domaine riche et mature, mais toujours actif. Parmi les sujets abordés par la recherche ces dernières années, on peut citer :

Compilateur certifié Prouver la correction d'un programme écrit en langage de haut niveau peut sembler dérisoire si la compilation de ce programme n'est pas correcte, c'est-à-dire qu'elle ne respecte pas la sémantique du langage. Par conséquent, des travaux ont été effectués sur la preuve formelle de correction d'un compilateur, par exemple le projet Compcert (`http://compcert.inria.fr/`).

Îlots formels On peut avoir envie d'étendre un langage sans en modifier profondément le noyau. Par exemple, on peut envisager d'ajouter du filtrage de motif (`match ... with`) à la ML) à un langage comme java. Le cadre des îlots formels consiste à étendre le langage de départ (langage océan) avec un langage îlot [Balland et al., 2006]. Un programme peut alors faire intervenir des expressions du langage îlots à l'intérieur d'expression du langage océan. Les expressions îlots sont alors dissoutes dans le langage océan, c'est-à-dire qu'elles sont compilées en expressions du langage océan. Si on reprend l'exemple du filtrage de motif en java, les constructions supplémentaires `match ... with` pourront être compilées en une succession de `if`.

Compilation de preuves De plus en plus de programmes informatiques sont vérifiés formellement. Dans les preuves de correction de ces programmes interviennent de façon assez intensive des calculs. Les démonstrations formelles de certains énoncés mathématiques, comme celle du théorème des quatre couleurs, font également intervenir de gros calculs. Pour être capable de construire et de vérifier de telles démonstrations, il est indispensable de rendre ce calcul efficace. Dans l'outil de démonstration Coq (`http://coq.inria.fr/`), à l'origine, le calcul dans les preuves était interprété. La possibilité de le compiler vers du bytecode interprété par un machine virtuelle a ensuite été rajoutée, et de récents travaux permettent de le compiler directement vers du code natif.

13. Références

Polycopiés de cours et tutoriaux

1. Cours de compilation de Tanguy Risset, http://perso.citi.insa-lyon.fr/trisset/cours/compil2004.html
2. Cours de compilation de François Pottier, http://www.enseignement.polytechnique.fr/informatique/INF564/
3. *Du langage à l'action : compilation et typage*, tutoriel de Xavier Leroy pour le cours de Gérard Berry au collège de France, http://pauillac.inria.fr/~xleroy/talks/compilation_typage_College_de_France.pdf.

Livres

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison Wesley, 1986. Le livre du dragon, appelé ainsi en raison de sa couverture. Déjà très complet, il existe une seconde édition parue en 2006 introduisant de nouveaux aspects. Les deux versions possèdent des traductions françaises.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. Plutôt bien écrit, existe aussi en version pour java et C.

Articles

Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33 :17–20, 1998.

Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal islands. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65. Springer, 2006.

Michel Schinz and Martin Odersky. Tail call elimination on the java virtual machine. In *Proc. ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability.*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 155–168. Elsevier, 2001.

A. Annexe

A.1. Expressions régulières de type lex

Par souci de simplicité, on ne donne ici que les expressions communes entre lex et ocamllex. Les caractères spéciaux de lex sont `" \ [ ] ^ - ? . * + | ( ) $ / { } % < >`. Un caractère non-spécial reconnaît le caractère en question. Pour reconnaître un caractère spécial, il faut le protèger par `\` . Une séquence de caractères entre `"` est reconnue telle quelle (c'est-à-dire qu'il n'y a pas besoin de protèger les caractères spéciaux à l'intérieur).

Les expressions sont construites de la façon suivante :

expr.	signification
c	reconnaît le caractère non-spécial c (en ocamllex, `'c'`)
\c	reconnaît le caractère spécial c (en ocamllex `'\c'`)
ef	reconnaît l'expression e puis l'expression f
"s"	reconnaît la chaîne de caractères s
.	reconnaît n'importe quel caractère (en ocamllex `_`)
[S]	reconnaît l'ensemble de caractère S, où S est une concaténation de caractères ou d'intervalles de caractères (par exemple `a-z`)
[^S]	reconnaît l'ensemble des caractères n'appartenant pas à S
e?	reconnaît optionnellement l'expression e
e+	reconnaît l'expression e une ou plusieurs fois
e*	reconnaît l'expression e zéro, une ou plusieurs fois
e|f	reconnaît l'expression e ou l'expression f
(e)	reconnaît e (utile pour changer la priorité des opérateurs)

Exemple A.1 *:* `[a-zA-Z_][a-zA-Z0-9_']*` reconnaît un identifiant ocaml. `\-?[1-9][0-9]*` reconnaît un entier relatif. `"/*"([^*]|\*[^/])*"*/"` reconnaît un commentaire C (rappelons que ceux-ci ne peuvent pas être imbriqués).

A.2. Calcul de points fixes

Définition A.1 (Treillis complet)**.** *Un* treillis *est un ensemble ordonné* $(E, \sqsupseteq)$ *tel que pour tous* a *et* b *dans* E*, l'ensemble* $\{a, b\}$ *possède une borne inférieure et une borne supérieure, notées respectivement* $a \sqcap b$ *et* $a \sqcup b$*.*

Un treillis est dit complet *si tout sous-ensemble* $A \subseteq E$ *admet une borne inférieure et une borne supérieure, notées respectivement* $\bigsqcap A$ *et* $\bigsqcup A$*.*

Exemple A.2 *:* L'ensemble des booléens avec $E = \{\bot, \top\}$ et $\top \sqsupset \bot$ est un treillis complet.

Pour tout ensemble A, l'ensemble des parties $\mathcal{P}(A)$ muni de l'inclusion $\supseteq$ est un treillis complet.

L'ensemble de entiers naturels $\mathbb{N}$ muni de la relation de divisibilité ($x|y$ si il existe un $z \in \mathbb{N}$ tel que $x \times z = y$) est une treillis complet dont les opérations $\sqcap$ et $\sqcup$ sont le pgcd et le ppcm.

Étant donné un ensemble quelconque D et un treillis complet $(E, \sqsupseteq)$, l'ensemble des fonctions de D dans E est un treillis complet pour l'ordre point-à-point : $f \sqsupseteq g$ ssi pour tout $x \in D$, $f(x) \sqsupseteq g(x)$. En particulier on a $f \sqcap g = \begin{cases} D \to E \\ x \mapsto f(x) \sqcap g(x) \end{cases}$.

Définition A.2 (Fonction monotone)**.** *Une fonction $f : E \to E$ sur un treillis complet est dite* monotone *si :*

pour tout $x, y \in E$, le fait que $x \sqsupseteq y$ implique $f(x) \sqsupseteq f(y)$.

Exemple A.3 *:* La fonction carré : $x \mapsto x^2$ sur $(\mathbb{N}, |)$ est monotone.

Étant donné un graphe (E, V), on définit la fonction suivante sur les fonctions de E dans $\mathcal{P}(E)$:

$$\text{étend} : \begin{cases} \mathcal{P}(E)^E \to \mathcal{P}(E)^E \\ \qquad f \mapsto x \mapsto \{x\} \cup \bigcup\limits_{(x,y) \in V} f(y) \end{cases}$$

On peut vérifier que cette fonction est monotone pour l'ordre point-à-point dans $\mathcal{P}(E)^E$.

Définition A.3 (Point fixe)**.** *Étant donnée une fonction $f : E \to E$, un élément $x \in E$ est appelé* point-fixe *de f si $f(x) = x$.*

Théorème A.1 (Knaster-Tarski)**.**
Étant donnés un treillis complet $(E, \sqsupseteq)$ et une fonction monotone $f : E \to E$ sur ce treillis, l'ensemble des points fixes de f forme un treillis complet.

En particulier, ce théorème implique l'existence d'un plus petit point fixe $lfp(f) \stackrel{\text{déf}}{=} \bigsqcap\{x \in E \mid f(x) = x\}$ et d'un plus grand point fixe $gfp(f) \stackrel{\text{déf}}{=} \bigsqcup\{x \in E \mid f(x) = x\}$. Quand E est fini, il est possible de calculer ces points fixes à l'aide d'un processus itératif. L'idée est de partir du plus petit élément de E, c'est-à-dire $\bot \stackrel{\text{déf}}{=} \bigsqcap E$, et d'appliquer f jusqu'à obtenir un point fixe. En effet, il est facile de montrer par récurrence que pour tout $n \in \mathbb{N}$, on a $f^{n+1}(\bot) \sqsupseteq f^n(\bot)$, mais puisque E est fini, on atteindra forcément un point où $f^{n+1}(\bot) = f^n(\bot)$. On peut alors montrer que le point fixe ainsi obtenu est bien le plus petit. Pour obtenir le plus grand, il suffit de partir de $\top \stackrel{\text{déf}}{=} \bigsqcup E$ au lieu de $\bot$. L'algorithme est donc le suivant :

```
r ← ⊥
faire
  r' ← f(r)
tant que r ≠ r'
retourner r
```

Exemple A.4 *:* On a *lfp*(carré) = 1 et *gfp*(carré) = 0. Ce sont les seuls points fixes de carré, l'ensemble de points fixes de carré est donc isomorphe aux booléens.

Le plus petit point fixe de étend est la relation d'accessibilité : $x \in \text{acc}(y)$ ssi il existe un chemin de x à y dans le graphe. En appliquant l'algorithme d'itération, au bout de n itérations on a calculé les nœuds accessibles en n pas. Le plus grand point fixe de étend n'est pas intéressant, il s'agit de la fonction qui à tout nœud associe l'ensemble des nœuds.

A.3. Instructions courantes RISC-V

Le processeur RISC-V est un processeur de type RISC (*Reduced Instruction Set Computing*) pour lequel les instructions, de taille fixe, sont régulières et où l'accès à la mémoire utilise des instructions spéciales. Ces processeurs possèdent de nombreux registres, lesquels sont uniformes. Il existe plusieurs simulateurs de machine RISC-V, dont Jupiter (`https://github.com/andrescv/Jupiter`). L'ensemble des instructions RISC-V est décrit dans le manuel du jeu d'instruction RISC-V disponible à l'adresse `https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf` . On peut également trouver sur Internet des fiches récapitulative, comme ici : `http://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf`. Nous nous contenterons ici de décrire les instructions les plus courantes de la version RV32IM.

Opérations sur les entiers

`add rd, rs, rt` ajoute les contenus des registres `rs` et `rt` et place le résultat dans le registre `rd`.

`addi rd, rs, i` ajoute la constante `i` au contenu du registre `rs` et place le résultat dans le registre `rd`.

`mul rd, rs, rt` multiplie les contenus des registres `rs` et `rt` et place le résultat dans le registre `rd`.

`sub rd, rs, rt` soustrait les contenus des registres `rs` et `rt` et place le résultat dans le registre `rd`.

`and rd, rs, rt` fait le « et » bit à bit entre les contenus des registres `rs` et `rt` et place le résultat dans le registre `rd`.

`slli rd, rs, i` décale les bits du contenu du registre `rs` de `i` positions vers la gauche, et place le résultat dans le registre `rd`.

`not rd, rs` fait la négation bit à bit du contenu du registre `rs`, et place le résultat dans le registre `rd`.

`li rd, i` place la constante `i` dans le registre `rd`.

`mv rd, rs` copie le contenu du registre `rs` dans le registre `rd`.

Comparaisons

`slt rd, rs, rt` met dans le registres `rd` la valeur 1 si le contenu de `rs` est plus petit que le contenu de `rt`, et 0 sinon.

`slti rd, rs, i` met dans le registres `rd` la valeur 1 si le contenu de `rs` est plus petit que la constante `i`, et 0 sinon.

`seq rd, rs, rt` met dans le registres `rd` la valeur 1 si le contenu de `rs` est égal au contenu de `rt`, et 0 sinon.

`sne rd, rs, rt` met dans le registres `rd` la valeur 1 si le contenu de `rs` n'est pas égal au contenu de `rt`, et 0 sinon.

Branchements et sauts

`bgez rs, label` continue à l'instruction indiquée par `label` si le contenu de `rs` est plus grand ou égal à zéro, continue normalement sinon. On a aussi `beq rs, rt, label` ; `bgtz rs, label` ; `blez rs, label` ; `bltz rs, label` ; etc.

`j i` saute à l'instruction `i`.

`jal i` saute à l'instruction `i` après avoir sauvegardé l'adresse de l'instruction suivante dans le registre `ra`.

`jr rs` saute à l'instruction dont l'adresse est le contenu de `rs`.

`ret` saute à l'instruction dont l'adresse est le contenu du registre `ra`.

Gestion de la mémoire

`lw rd, i(rt)` place dans le registre `rd` le contenu de la mémoire à l'adresse donnée par le contenu de `rt` plus `i`.

`sw rs, i(rt)` stocke le contenu du registre `rd` à l'emplacement mémoire donné par le contenu de `rt` plus `i`.

Appels systèmes

`ecall` appel système (environnement) ; le comportement dépend de l'environnement dans lequel le code est exécuté (système d'exploitation, superviseur, etc.).

Syntaxe abstraite de Pseudo-Pascal

François Pottier

Cette fiche récapitule la syntaxe abstraite du langage Pseudo-Pascal. Elle correspond à la définition contenue dans le fichier **PP.mli** du compilateur.

La méta-variable b parcourt l'ensemble des booléens, à savoir $\{\text{true}, \text{false}\}$. La méta-variable n parcourt l'ensemble des entiers munis d'un signe représentables sur 32 bits, à savoir l'intervalle $[-2^{31} \ldots 2^{31}-1]$. La méta-variable x parcourt un ensemble dénombrable d'identificateurs employés pour nommer des variables. La méta-variable f parcourt un ensemble dénombrable d'identificateurs employés pour nommer des procédures ou fonctions.

Les types sont integer, boolean, et array of τ, où τ est lui-même un type, et décrit les éléments du tableau.

τ	::=		**types**
	|	integer	entiers
	|	boolean	booléens
	|	array of τ	tableaux

Les constantes sont booléennes ou entières. Il n'y a pas de constantes de type tableau : les tableaux sont alloués dynamiquement.

k	::=		**constantes**
	|	b	constante booléenne
	|	n	constante entière

Les opérateurs unaires et binaires, qui apparaissent dans la syntaxe des expressions, sont les suivants. Tous s'appliquent à des expressions de type integer. Tous produisent un résultat de type integer, sauf les opérateurs binaires de comparaison, qui produisent un résultat de type boolean.

uop	::=		**opérateurs unaires**
	|	$-$	négation
bop	::=		**opérateurs binaires**
	|	$+$	addition
	|	$-$	soustraction
	|	$\times$	multiplication
	|	$/$	division
	|	$< \mid \leq \mid > \mid \geq \mid = \mid \neq$	comparaison

Les opérations primitives, c'est-à-dire les procédures et fonctions prédéfinies, sont les suivantes.

π	::=		**opérations primitives**
	|	write	affichage d'un entier
	|	writeln	affichage d'un entier et retour à la ligne
	|	readln	lecture d'un entier

La cible d'un appel de procédure ou fonction est soit une opération primitive, soit une procédure ou fonction définie par l'utilisateur, laquelle est alors désignée par son nom.

φ	::=		**cible d'un appel**
	|	π	opération primitive
	|	f	procédure ou fonction définie par l'utilisateur

La syntaxe des expressions, conditions, et instructions est la suivante. Notons que la distinction entre ces trois catégories est quelque peu arbitraire : on pourrait ne distinguer que deux catégories (expressions et instructions), comme en C, ou bien une seule (expressions), comme en Objective Caml. Les distinctions que la syntaxe du langage n'impose pas seront alors (si nécessaire) effectuées lors de la vérification des types.

e	::=		**expressions**
	|	k	constante
	|	x	variable
	|	uop e	application d'un opérateur unaire
	|	e bop e	application d'un opérateur binaire
	|	$\varphi(e \ldots e)$	appel de fonction
	|	$e[e]$	lecture dans un tableau
	|	new array of τ $[e]$	allocation d'un tableau
c	::=		**conditions**
	|	e	expression (à valeur booléenne)
	|	not c	négation
	|	c and c	conjonction
	|	c or c	disjonction
i	::=		**instructions**
	|	$\varphi(e \ldots e)$	appel de procédure
	|	$x := e$	affectation
	|	$e[e] := e$	écriture dans un tableau
	|	$i \ldots i$	séquence
	|	if c then i else i	conditionnelle
	|	while c do i	boucle

Une définition de procédure ou fonction est composée d'un en-tête, d'une série de déclarations de variables locales, et d'un corps. L'en-tête donne le nom de la procédure ou fonction, déclare ses paramètres formels et, s'il s'agit d'une fonction, indique le type de son résultat. J'écris informellement $\tau^?$ pour indiquer un type τ optionnel.

d	::=		**définitions de procédures/fonctions**
	|	$f(x : \tau \ldots x : \tau) : \tau^?$	en-tête
		var $x : \tau \ldots x : \tau$	variables locales
		i	corps

Un programme est composé d'une série de déclarations de variables globales, d'une série de déclarations de procédures ou fonctions, et d'un corps.

p	::=		**programme**
	|	var $x : \tau \ldots x : \tau$	variables globales
		$d \ldots d$	définitions de procédures/fonctions
		i	corps

Sémantique opérationnelle de Pseudo-Pascal

François Pottier

Cette fiche récapitule la sémantique opérationnelle structurée à grands pas du langage Pseudo-Pascal.

1 Valeurs

Les **valeurs** manipulées au cours de l'exécution sont des booléens, des entiers représentables sur 32 bits, des **adresses** de tableau, ou bien la constante nil. Un tableau est représenté par son adresse en mémoire, et non pas directement par la suite de ses éléments. Cette distinction est importante : elle signifie par exemple qu'il est possible pour deux variables distinctes de type « array of τ » d'être **alias** l'une de l'autre, c'est-à-dire de contenir la même adresse, donc de pointer vers le même tableau. Elle signifie également que le contenu d'une variable de type tableau peut être stocké dans un registre, puisqu'il s'agit d'une adresse et non du tableau lui-même. Le tableau lui-même sera stocké dans le tas. La valeur nil est la valeur attribuée aux variables de type tableau non encore initialisées.

v	::=		**valeurs**
		b	constante booléenne
	|	n	constante entière
	|	ℓ	adresse de tableau
	|	nil	adresse invalide

En Pseudo-Pascal, lorsqu'on déclare une variable, on donne son type τ, mais on ne spécifie pas quelle est sa valeur initiale. Dans certains langages, comme C, la valeur initiale de la variable est alors indéfinie : il est interdit d'accéder au contenu de cette variable avant de l'avoir initialisée. Dans d'autres, comme Java, la variable est considérée comme initialisée avec une valeur par défaut, laquelle dépend de τ. Pseudo-Pascal suit cette approche et définit une fonction qui à tout type τ associe une valeur par défaut de type τ :

$$\begin{aligned} \text{default(boolean)} &= \text{false} \\ \text{default(integer)} &= 0 \\ \text{default(array of } \tau\text{)} &= \text{nil} \end{aligned}$$

Le troisième cas explique pourquoi nous avons besoin de la valeur nil.

2 Opérateurs

La sémantique des opérateurs unaires est donnée par une fonction $[\![\cdot]\!]$ qui à tout opérateur uop associe une fonction partielle $[\![\text{uop}]\!]$ des valeurs dans les valeurs. Par exemple, la sémantique de l'opérateur de négation est donnée en définissant $[\![-]\!]$ comme la fonction qui à toute constante entière n associe la constante entière $-n$, normalisée à l'intervalle $[-2^{31} \ldots 2^{31} - 1]$. Il est important de noter que la fonction $[\![-]\!]$ est indéfinie en ce qui concerne les autres formes de valeurs : on ne peut l'appliquer ni à une constante booléenne, ni à une adresse, ni à la valeur nil. La sémantique d'un programme qui appliquerait l'opérateur de négation à un tableau,

par exemple, est donc indéfinie. Un tel programme est heureusement exclu par le système de types, puisque les types « integer » et « array of τ » sont incompatibles.

De même, la sémantique des opérateurs binaires est donnée par une fonction $[\![\cdot]\!]$ qui à tout opérateur bop associe une fonction partielle $[\![\text{bop}]\!]$ des paires de valeurs dans les valeurs. La définition exacte de cette fonction pour les quatre opérateurs arithmétiques et les six opérateurs de comparaison de Pseudo-Pascal est laissée en exercice au lecteur !

3 Jugements

Le contenu des variables globales est donné par un **environnement** G qui aux variables associe des valeurs. De même, le contenu des variables locales est donné par un environnement E. Enfin, le **tas** H associe aux adresses des suites finies de valeurs.

La sémantique de Pseudo-Pascal est définie par des **jugements**. L'évaluation d'une expression, d'une condition ou d'une instruction peut modifier les variables globales, les variables locales, ainsi que le tas. Les jugements qui décrivent l'évaluation mentionnent donc d'une part l'état initial G, H, E, d'autre part l'état final G', H', E'. Dans le cas des expressions et des conditions, ces jugements mentionnent également une valeur, qui représente le résultat de l'évaluation. Nos jugements sont donc de la forme

$$\begin{aligned} G, H, E/e &\longrightarrow G', H', E'/v \\ G, H, E/c &\longrightarrow G', H', E'/b \\ G, H, E/i &\longrightarrow G', H', E' \end{aligned}$$

Le premier de ces jugements se lit : « dans l'état décrit par G, H, E, l'évaluation de l'expression e mène à l'état décrit par G', H', E' et produit la valeur v ». Les second et troisième jugements se lisent de façon similaire. Nous écrirons parfois S pour représenter le triplet G, H, E.

Ce style de sémantique est dit « à grands pas » car ces jugements relient directement expressions et résultats, sans mentionner explicitement les étapes intermédiaires du calcul.

La sémantique des expressions est donnée par la figure 1. Celle-ci contient un ensemble de règles qui définissent le jugement $G, H, E/e \longrightarrow G', H', E'/v$. De même, les jugements qui décrivent la sémantique des conditions et des instructions sont définis par les règles des figures 2 et 3. Par abus de notation, toutes ces règles sont implicitement paramétrées par une série de définitions de fonctions et procédures, qui sont consultées par les deux règles qui décrivent l'appel à une fonction ou procédure définie par l'utilisateur. Ces définitions de fonctions et procédures sont fixées lorsqu'on s'intéresse à un programme p précis.

La sémantique d'un programme p est définie par un dernier jugement, de la forme

$$p \longrightarrow$$

Le jugement $p \longrightarrow$ se lit « Le programme p s'exécute sans erreur et termine ». Sa définition est donnée par la figure 4.

Constante
$$S/k \longrightarrow S/k$$

Variable locale
$$\frac{x \in dom(E)}{G,H,E/x \longrightarrow G,H,E/E(x)}$$

Variable globale
$$\frac{x \in dom(G) \setminus dom(E)}{G,H,E/x \longrightarrow G,H,E/G(x)}$$

Opérateur unaire
$$\frac{S/e \longrightarrow S'/v}{S/\mathrm{uop}\ e \longrightarrow S'/[\![\mathrm{uop}]\!](v)}$$

Opérateur binaire
$$\frac{S/e_1 \longrightarrow S'/v_1 \qquad S'/e_2 \longrightarrow S''/v_2}{S/e_1\ \mathrm{bop}\ e_2 \longrightarrow S''/[\![\mathrm{bop}]\!](v_1,v_2)}$$

Évaluation des arguments de fonction
$$\frac{\forall j \in \{1\ \ldots\ n\} \quad S_{j-1}/e_j \longrightarrow S_j/v_j \qquad S_n/\varphi(v_1\ \ldots\ v_n) \longrightarrow S'/v}{S_0/\varphi(e_1\ \ldots\ e_n) \longrightarrow S'/v}$$

Appel de readln
$$S/\mathrm{readln}() \longrightarrow S/n$$

Appel d'une fonction définie
$$\frac{\begin{array}{c} p \ni f(x_1:\tau_1\ \ldots\ x_n:\tau_n):\tau \quad \mathrm{var}\ x'_1:\tau'_1\ \ldots\ x'_q:\tau'_q \quad i \\ E' = (x_j \mapsto v_j)_{1\le j\le n} \cup (x'_j \mapsto \mathrm{default}(\tau'_j))_{1\le j\le q} \cup (f \mapsto \mathrm{default}(\tau)) \\ G,H,E'/i \longrightarrow G',H',E'' \qquad v = E''(f) \end{array}}{G,H,E/f(v_1\ \ldots\ v_n) \longrightarrow G',H',E/v}$$

Lecture dans un tableau
$$\frac{\begin{array}{c} S/e_1 \longrightarrow S'/\ell \qquad S'/e_2 \longrightarrow S''/n \\ S'' = G'',H'',E'' \qquad H''(\ell) = v_0 \ldots v_{p-1} \qquad 0 \le n < p \end{array}}{S/e_1[e_2] \longrightarrow S''/v_n}$$

Allocation d'un tableau
$$\frac{\begin{array}{c} S/e \longrightarrow G',H',E'/n \qquad n \ge 0 \\ \ell \# H' \qquad H'' = H'[\ell \mapsto \mathrm{default}(\tau)^n] \end{array}}{S/\mathrm{new\ array\ of}\ \tau\ [e] \longrightarrow G',H'',E'/\ell}$$

Fig. 1 — Sémantique des expressions

Expression à valeur booléenne
$$\frac{S/e \longrightarrow S'/b}{S/e \longrightarrow S'/b}$$

Négation
$$\frac{S/c \longrightarrow S'/b}{S/\mathrm{not}\ c \longrightarrow S'/\neg b}$$

Conjonction (si)
$$\frac{S/c_1 \longrightarrow S'/\mathrm{false}}{S/c_1\ \mathrm{and}\ c_2 \longrightarrow S'/\mathrm{false}}$$

Conjonction (sinon)
$$\frac{\begin{array}{c} S/c_1 \longrightarrow S'/\mathrm{true} \\ S'/c_2 \longrightarrow S''/b \end{array}}{S/c_1\ \mathrm{and}\ c_2 \longrightarrow S''/b}$$

Disjonction (si)
$$\frac{S/c_1 \longrightarrow S'/\mathrm{true}}{S/c_1\ \mathrm{or}\ c_2 \longrightarrow S'/\mathrm{true}}$$

Disjonction (sinon)
$$\frac{\begin{array}{c} S/c_1 \longrightarrow S'/\mathrm{false} \\ S'/c_2 \longrightarrow S''/b \end{array}}{S/c_1\ \mathrm{or}\ c_2 \longrightarrow S''/b}$$

Fig. 2 — Sémantique des conditions

Évaluation des arguments d'une procédure

$$\frac{\forall j \in \{1 \ldots n\} \quad S_{j-1}/e_j \longrightarrow S_j/v_j \qquad S_n/\varphi(v_1 \ \ldots \ v_n) \longrightarrow S'}{S_0/\varphi(e_1 \ \ldots \ e_n) \longrightarrow S'}$$

Appel de write

$$S/\mathsf{write}(n) \longrightarrow S$$

Appel de writeln

$$S/\mathsf{writeln}(n) \longrightarrow S$$

Appel d'une procédure définie

$$\frac{p \ni f(x_1 : \tau_1 \ \ldots \ x_n : \tau_n) \quad \mathsf{var}\ x'_1 : \tau'_1 \ \ldots \ x'_q : \tau'_q \quad i \qquad E' = (x_j \mapsto v_j)_{1 \le j \le n} \cup (x'_j \mapsto \mathrm{default}(\tau'_j))_{1 \le j \le q} \qquad G, H, E'/i \longrightarrow G', H', E''}{G, H, E/f(v_1 \ \ldots \ v_n) \longrightarrow G', H', E}$$

Affectation: variable locale

$$\frac{S/e \longrightarrow G', H', E'/v \qquad x \in \mathrm{dom}(E')}{S/x := e \longrightarrow G', H', E'[x \mapsto v]}$$

Affectation: variable globale

$$\frac{S/e \longrightarrow G', H', E'/v \qquad x \in \mathrm{dom}(G') \setminus \mathrm{dom}(E')}{S/x := e \longrightarrow G'[x \mapsto v], H', E'}$$

Écriture dans un tableau

$$\frac{S/e_1 \longrightarrow S'/\ell \qquad S'/e_2 \longrightarrow S''/n \qquad S''/e_3 \longrightarrow G''', H''', E'''/v \qquad H'''(\ell) = v_0 \ldots v_{p-1} \qquad 0 \le n < p}{S/e_1[e_2] := e_3 \longrightarrow G''', H'''[\ell \mapsto v_0 \ldots v_{n-1}\, v\, v_{n+1} \ldots v_{p-1}], E'''}$$

Séquence

$$\frac{\forall j \in \{1 \ldots n\} \quad S_{j-1}/i_j \longrightarrow S_j}{S_0/i_1 \ldots i_n \longrightarrow S_n}$$

Conditionnelle (si)

$$\frac{S/c \longrightarrow S'/\mathsf{true} \qquad S'/i_1 \longrightarrow S''}{S/\mathsf{if}\ c\ \mathsf{then}\ i_1\ \mathsf{else}\ i_2 \longrightarrow S''}$$

Conditionnelle (sinon)

$$\frac{S/c \longrightarrow S'/\mathsf{false} \qquad S'/i_2 \longrightarrow S''}{S/\mathsf{if}\ c\ \mathsf{then}\ i_1\ \mathsf{else}\ i_2 \longrightarrow S''}$$

Boucle (si)

$$\frac{S/c \longrightarrow S'/\mathsf{true} \qquad S'/i; \mathsf{while}\ c\ \mathsf{do}\ i \longrightarrow S''}{S/\mathsf{while}\ c\ \mathsf{do}\ i \longrightarrow S''}$$

Boucle (sinon)

$$\frac{S/c \longrightarrow S'/\mathsf{false}}{S/\mathsf{while}\ c\ \mathsf{do}\ i \longrightarrow S'}$$

Fig. 3 — Sémantique des instructions

Programme

$$\frac{G = (x_j \mapsto \mathrm{default}(\tau_j))_{1 \le j \le n} \qquad G, \emptyset, \emptyset/i \longrightarrow S'}{\mathsf{var}\ x_1 : \tau_1 \ \ldots \ x_n : \tau_n \quad d \ \ldots \ d \quad i \longrightarrow}$$

Fig. 4 — Sémantique des programmes