IFIPS – Université Paris Sud
Cycle ingénieur 2^{e} année Informatique

Cours de Compilation

Christine Paulin-Mohring, Marc Pouzet
et Jean-Christophe Filliâtre

Préambule

Ce cours présente les principales phases de la construction d'un compilateur, c'est-à-dire un *programme* qui transforme une *suite de caractères* représentant un *programme* en une suite *d'instructions machine* qui pourront s'exécuter pour produire le *résultat* du programme.

Un compilateur met en œuvre des méthodes et outils d'*analyse lexicale et syntaxique* ; l'apprentissage du fonctionnement de ces analyses constitue une partie importante du cours. Les langages de programmation modernes de haut niveau proposent une détection précoce des erreurs grace à une *analyse sémantique* souvent présente sous la forme d'un contrôle de types. La dernière phase de la compilation est la *génération de code* qui se fait en plusieurs étapes correspondant à différents langages intermédiaires avant d'aboutir au code exécutable de la machine. Nous étudierons plus particulièrement l'organisation de la mémoire pour la gestion des appels de procédures.

À travers l'étude de la compilation, nous chercherons à comprendre les caractéristiques des langages de programmation. Un compilateur est un programme de taille importante qu'il est nécessaire de bien structurer, il doit également être efficace et de plus il est important qu'il soit exempt d'erreurs ; être capable d'écrire un bon compilateur est donc un défi pour tout programmeur chevronné.

Quelques informations utiles

La page WEB du cours : `http://www.lri.fr/~filliatr/ifips/`

Évaluation : Le module comporte un examen final. Le cours est associé à un projet (travaux pratiques) au cours duquel un petit compilateur sera réalisé.

Coordonnées :

Adresse électronique `Jean-Christophe.Filliatre@lri.fr`

Merci d'utiliser prioritairement le courrier électronique.

Table des matières

Chapitre 1

Introduction à la compilation

1.1 Rappels de notation

On appellera *alphabet* un ensemble fini dont les éléments seront appelés *lettres*.
Un *mot* sur un alphabet A est une suite finie d'éléments de A. Un mot de longueur n composé des lettres $a_1, \ldots, a_n$ sera noté $a_1 a_2 \ldots a_n$. Le mot vide est noté ϵ. La concaténation de deux mots w_1 et w_2 est notée $w_1 w_2$.
L'ensemble des mots sur A est noté A^* (monoide libre).

Un *langage* sur un alphabet A est un ensemble de mots de A i.e. une partie de A^*.

1.2 Qu'est-ce que la compilation ?

1.2.1 À quoi cela sert ?

Un *traducteur* est un programme qui transforme chaque mot d'un langage L_1 sur un alphabet A_1 en un mot d'un langage L_2 sur un alphabet A_2.

Un *compilateur* est un cas particulier de traducteur qui prend en entrée un mot représentant un programme c'est-à-dire la description d'un calcul et qui est chargé de produire le résultat de ce calcul. En pratique, il s'agira de traduire un texte représentant le calcul vers une suite d'instructions exécutables par la machine.

Exemples

- D'un langage de haut niveau (Pascal, Fortran, C++, ML, ...) vers un langage d'assembleur.
- Du langage d'assembleur vers du code binaire.
- D'un langage de haut niveau vers les instructions d'une machine virtuelle (JVM).
- D'un langage de description de texte (HTML, Latex, Word) vers le langage postscript qui est lui-même traduit en commandes d'impression pour une imprimante.
- Traitement de langages spécialisés : base de données (SQL), langages réactifs (Esterel, Lustre) circuits, langages de macros
- Compilation vers des processeurs spécialisés (téléphones mobiles, téléviseurs,robots)
- ...

Une des techniques mises en œuvre, l'analyse syntaxique, est également utile pour le traitement des entrées d'un programme : d'une chaîne de caractères vers des données structurées utilisées par les procédures internes du programme.

1.2.2 Les difficultés

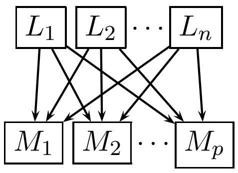
- Un compilateur est un programme complexe qui nécessite en général de passer par de nombreuses structures intermédiaires :
  Chaîne de caractères, liste de "mots", arbre de syntaxe abstraite, code intermédiaire. Comme dans beaucoup de problèmes informatiques, il sera essentiel de choisir les bonnes structures pour une application particulière qui seront souvent des compromis entre les impératifs de clarté, d'efficacité en temps et en espace.
- Un compilateur traite souvent des programmes incorrects, il faut détecter les erreurs et faire un diagnostic correct pour l'utilisateur.
- Un compilateur doit permettre d'exécuter efficacement les calculs.

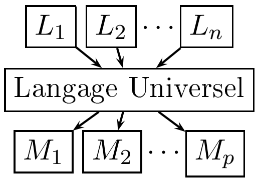
1.2.3 Les méthodes

On distingue une phase *d'analyse* qui reconnaît qu'une chaîne de caractères est la description correcte d'un calcul. L'analyse construit à partir de l'entrée d'une chaîne de caractères, une donnée structurée (appelée arbre de syntaxe abstraite). Cette analyse est largement indépendante du langage cible de la compilation.

Vient ensuite une phase de *synthèse* qui, à partir de cette donnée structurée va construire le résultat c'est-à-dire une description du même calcul dans un langage plus proche du langage machine.

La compilation de n langages pour p architectures nécessite a priori la réalisation de $n \times p$ compilateurs. En utilisant un langage universel intermédiaire on réduit le problème à la construction de $n + p$ transformations.





1.2.4 Un exemple d'analyse

L'exemple suivant illustre le passage d'une expression arithmétique simple, vue comme une suite de caractères, à une expression sous forme d'arbre.

Si on veut calculer la "valeur" d'une expression arithmétique :

$$x1 + (y - 41 - z) * -t$$

il faut d'abord reconnaître les différentes entités. Les deux caractères x et 1 représentent une entité à savoir un identificateur de valeur $x1$. Les deux caractères 4 et 1 juxtaposés représentent une entité à savoir une constante entière de valeur 41. Le signe $-$ est interprété comme un symbole spécial représentant la soustraction. Si le même signe était autorisé dans les identificateurs il y aurait ambiguïté et il faudrait choisir entre l'identificateur `y-41` et l'expression arithmétique $\mathtt{y} - 41$.

Ces règles qui nous paraissent naturelles sont en fait liées à des conventions qu'il est essentiel d'expliciter pour chaque langage à implanter.

Le calcul ne peut s'effectuer qu'en connaissant la chaîne complète et les règles implicites :
- Comment les opérateurs associent : $(y - 41) - z$ ou $y - (41 - z)$
- Les précédences : $x1 + ((y - 41 - z) * -t)$ ou $(x1 + (y - 41 - z)) * -t$

On peut adopter une notation linéaire non ambiguë mais beaucoup plus lourde :

$$x1 + (((y - 41) - z) * -t)$$

ou bien avoir recours à une notation arborescente. Les feuilles de ces arbres sont formées des identificateurs ou des constantes. Les nœuds sont soit binaires dans le cas de $+$, $-$ ou $*$ soit unaire dans le cas de l'opposé ($-$ unaire). On parle alors d'*arbres de syntaxe abstraite*. Beaucoup d'opérations se construisent aisément comme des procédures récursives par rapport à la structure de l'arbre représentant une expression.

Remarques

- Le choix d'une bonne structure d'arbre de syntaxe abstraite est essentiel pour la phase d'analyse sémantique et de synthèse. Il représente de manière non-ambiguë et non redondante, la structure du calcul à effectuer.
  Les arbres de syntaxe abstraite et les expressions arithmétiques bien formées sont en correspondance bijective. Ce n'est pas vrai pour l'ensemble des mots construits sur l'alphabet $\{x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, -, (,)\}$.
- La construction de l'arbre de syntaxe abstraite se fera par un calcul d'attributs associé à la grammaire qui reconnaît le langage.
- Les langages de haut niveau utilisent des identificateurs pour représenter des cases mémoires, les langages machines utilisent par contre des adresses. Les identificateurs peuvent être des mots longs qui nécessitent beaucoup de place mémoire. Il est donc justifié de les remplacer tôt par un entier afin de partager les différentes instances et de tester rapidement l'égalité. La *table de symboles* permet de garder une correspondance entre un nom connu de l'utilisateur et une représentation interne efficace.

Dans certains cas simples, les étapes d'analyse et de synthèse peuvent être combinées et le code cible engendré en une seule passe sans construire la structure intermédiaire d'arbre de syntaxe abstraite.

1.2.5 Qu'attend-on d'un compilateur ?

La détection des erreurs

Un compilateur doit pouvoir détecter les erreurs *statiques* qui ne nécessitent pas l'exécution du programme et être capable de reporter cette erreur à l'utilisateur.

Exemple

- Identificateurs mal formés
- Constructions syntaxiques incorrectes
- Expressions mal typées `if 3 then "toto" else 4.5`
- Références non instanciées
- ...

Les erreurs détectées à l'exécution s'appellent les erreurs *dynamiques* il s'agit par exemple de la division par zéro ou du dépassement des bornes dans un tableau.

L'efficacité

Le programme de compilation doit être si possible rapide (en particulier ne pas boucler) et produire un code qui s'exécutera rapidement.

La correction

Un compilateur doit être correct, c'est-à-dire que le calcul décrit dans le langage de haut niveau doit être effectivement celui exécuté par la machine. Dans la pratique, la sémantique des langages de programmation est très souvent incomplètement spécifiée. Cette situation pose des problèmes pour assurer la correction des applications (code mobile Java). Le résultat d'un même programme pourra donc varier suivant le compilateur utilisé ...

La modularité

Lorsque l'on construit un gros développement, il est important de pouvoir profiter de la *compilation séparée*. Chaque module peut être compilé sans connaître l'implantation de toutes les fonctions qu'il utilise. On se contente en général d'une *signature* donnant par exemple les indications de typage des opérations. Le même programme compilé pourra être utilisé dans des contextes différents.

1.2.6 Quelques notions de sémantique

À un langage de programmation est en général associé une sémantique qui décrit le "sens" du calcul. L'effet du calcul est une modification de l'état de la machine. L'état de la mémoire peut être modélisé comme une fonction de l'ensemble des places mémoires (adresses) vers les valeurs stockées. Nous notons Mem l'ensemble des états de la mémoire. Si $m \in \mathsf{Mem}$ et i est une adresse alors $m(i)$ représente la valeur de la mémoire à l'adresse i.

Environnement et état

Un programme manipule des variables, celles-ci servent à représenter des zones de mémoire dans lesquelles on peut stocker des valeurs. Pour décrire le comportement d'un programme, on distingue deux composantes :

l'environnement associe une place dans la mémoire à chaque déclaration de variables. Si le programme ne contient que des variables globales, ne manipule pas de pointeurs et ne passe les paramètres des procédures que par valeur, alors l'environnement est complètement déterminé à la compilation, à chaque déclaration de variable correspond exactement un emplacement mémoire. En présence de pointeurs, deux déclarations de variable peuvent correspondre à la même position dans la mémoire, et l'environnement peut être modifié au cours du calcul.

l'état associe une valeur à chaque emplacement de la mémoire (éventuellement une valeur non définie si la mémoire n'est pas initialisée).

Exemple

On considère un langage où l'environnement n'est pas modifié par le calcul. On s'intéressera aux transformations de l'état de la mémoire.

Il y a plusieurs formes de sémantique que nous illustrerons sur l'exemple suivant :

```
x :=1 ; y :=x+1+y ; skip
```

On suppose que notre langage permet de manipuler deux variables x et y, des constantes entières n et des expressions entières formées à l'aide du signe d'addition.

$$e ::= x \mid y \mid n \mid e + e$$

Une instruction simple est soit l'affectation d'une expression à une variable, soit l'instruction `skip`, un programme est une suite d'instructions séparées par des `;`.

$$p ::= i \mid i\,;p \qquad i ::= x\ \texttt{:=e} \mid y\ \texttt{:=e} \mid \texttt{skip}$$

On notera $\rho(x)$ et $\rho(y)$ les adresses associées à x et y et on supposera $\rho(x) \neq \rho(y)$.

Sémantique dénotationnelle

En sémantique dénotationnelle, un programme est représenté comme une fonction mathématique f de domaine et image Mem. Soit m une mémoire, $f(m)$ est une mémoire c'est-à-dire une fonction des adresses dans les valeurs.

Pour notre exemple :

$$f(m)(\rho(x)) = 1, f(m)(\rho(y)) = m(\rho(y)) + 2, f(m)(i) = m(i)\ i \neq \rho(x), \rho(y)$$

Sémantique naturelle

La sémantique naturelle définit (en général de manière récursive par rapport au programme) une relation entre un état de mémoire m_1, un programme p et la mémoire finale obtenue après exécution du programme p. On distingue la sémantique des expressions et celle des programmes. On note $m \vdash e \rightsquigarrow v$ la relation qui dit que l'expression e s'évalue dans la mémoire m en la valeur v et $m_1 \vdash p \rightsquigarrow m_2$ la relation qui dit que l'expression p s'exécute dans une mémoire m_1 et transforme la mémoire en m_2. Pour notre exemple :

$$m \vdash x \rightsquigarrow m(\rho(x)) \qquad m \vdash y \rightsquigarrow m(\rho(y)) \qquad m \vdash 1 \rightsquigarrow 1$$

$$\frac{m \vdash t \rightsquigarrow v \qquad m \vdash u \rightsquigarrow w}{m \vdash t + u \rightsquigarrow v + w}$$

$$m \vdash \texttt{skip} \rightsquigarrow m \qquad \frac{m_1 \vdash p_1 \rightsquigarrow m \quad m \vdash p_2 \rightsquigarrow m_2}{m_1 \vdash p_1; p_2 \rightsquigarrow m_2}$$

$$\frac{m \vdash e \rightsquigarrow v \qquad\qquad m'(\rho(z)) = e, m'(i) = m(i)\ i \neq \rho(z)}{m \vdash z := e \rightsquigarrow m'}$$

On ne parle que des programmes qui terminent. Certains aspects de l'exécution (par exemple l'ordre d'évaluation des arguments) ne sont pas spécifiés dans cette sémantique.

Sémantique opérationnelle à petit pas

La sémantique opérationnelle décrit les transformations successives de la mémoire et du programme. L'évaluation d'une expression est décrite en terme de transformation de l'expression jusqu'à obtention d'une valeur qui ne peut plus être transformée. On note $(m|p)$ le programme p dans la mémoire m. La valeur d'un programme est obtenue lorsque plus aucune transformation ne s'applique. Les valeurs sont vues comme des expressions particulières de programme notées v, w.

$$m \vdash x \rightsquigarrow m(\rho(x)) \qquad m \vdash y \rightsquigarrow m(\rho(y))$$

$$\frac{m \vdash t \rightsquigarrow t'}{m \vdash t + u \rightsquigarrow t' + u} \qquad \frac{m \vdash u \rightsquigarrow u'}{m \vdash v + u \rightsquigarrow v + u'} \qquad \frac{v + w = r}{m \vdash v + w \rightsquigarrow r}$$

$$\frac{m \vdash t \rightsquigarrow t'}{(m|z := t; p) \rightsquigarrow (m|z := t'; p)} \qquad \frac{m'(\rho(z)) = v \quad [m'(i) = m(i)\ \forall i \neq \rho(z)]}{(m|z := v; p) \rightsquigarrow (m'|p)}$$

$$\frac{m'(\rho(z)) = v \quad [m'(i) = m(i)\ \forall i \neq \rho(z)]}{(m|z := v) \rightsquigarrow (m'|\texttt{skip})}$$

Correction d'un compilateur

Pour spécifier la correction d'un compilateur les sémantiques naturelles ou opérationnelles sont les plus utiles. Une compilation de L_1 vers L_2 sera correcte si étant données deux définitions de la sémantique des programmes de L_1 et de L_2, pour tout programme p de L_1, la sémantique de p est équivalente à la sémantique de la version compilée de p dans L_2.

$$\begin{array}{rcl} L_1 & \stackrel{\text{compilation}}{\longrightarrow} & L_2 \\ \text{sémantique} \downarrow & & \downarrow \text{sémantique} \\ V_1 & \longleftrightarrow & V_2 \end{array}$$

1.3 Les différentes phases de la compilation

1.3.1 Analyse

La phase d'analyse correspond à reconnaître qu'une entrée est un programme correct du langage source. Cette analyse doit être la plus rapide possible. C'est pourquoi on la structure en trois parties (analyse lexicale, analyse syntaxique, analyse sémantique) chaque phase utilise des outils de plus en plus complexes permettant d'avoir une compréhension de plus en plus fine du sens du programme.

Analyse lexicale

Il s'agit de lire la suite de caractères un à un et de séparer les *unités lexicales*.
- Séparateurs, commentaires, mots clés.
- Identificateurs, entiers, flottants, symboles ou suites de symboles.
- Directives de compilation.

Ces entités correspondent à des expressions régulières. Certaines ambiguïtés peuvent apparaître : >= peut correspondre à une ou deux unités lexicales.

Le *crible* permet d'exécuter des actions au moment de la reconnaissance de chaque unité lexicale. Cette phase permet de traiter et de classifier les unités. On pourra par exemple séparer les mots clés des identificateurs.

Il y a a priori une infinité d'unités lexicales reconnaissables, pourtant la grammaire ne peut traiter qu'un alphabet fini. Heureusement la grammaire n'a besoin de distinguer que des classes d'unités lexicales (identificateurs, valeurs numériques, commentaires, ...). On distingue donc pour chaque unité lexicale, sa classe (aussi appelée *token*) et sa valeur qui sera utilisée dans la construction de l'arbre de syntaxe abstraite.

À la fin de l'analyse lexicale, on a une suite de tokens associés à des valeurs.

Des programmes tels que lex, jflex, ocamllex, etc., permettent d'engendrer automatiquement des analyseurs lexicaux à partir d'une description des classes d'unités lexicales sous forme d'expressions régulières.

Analyse syntaxique

Il s'agit de reconnaître la structure du programme. L'analyse syntaxique détecte que le programme donné en entrée satisfait les règles grammaticales du langage utilisé.

L'analyse syntaxique peut se faire suivant des *méthodes ascendantes ou descendantes*.

Le langage devra être descriptible par une grammaire ayant de bonnes propriétés pour obtenir une analyse efficace. Des programmes tels que yacc, bison, cup, ocamlyacc, etc., permettent d'engendrer automatiquement des analyseurs syntaxiques à partir d'une description de la grammaire du langage lorsque celle-ci a de bonnes propriétés.

Les *actions* associées aux règles de grammaire permettent de construire un arbre de syntaxe abstraite comme résultat de l'analyse syntaxique.

Analyse sémantique

Il s'agit d'analyser l'arbre de syntaxe abstraite pour vérifier des propriétés dites statiques telles que la bonne utilisation des variables, le typage correct des expressions.

Si les opérateurs arithmétiques sont surchargés comme les opérations arithmétiques qui peuvent s'appliquer aux entiers et aux flottants, la phase d'analyse sémantique permettra de résoudre cette ambiguïté et d'appeler les opérations machines adaptées.

On peut encore utiliser des techniques d'analyse de flots de données ou d'interprétation abstraite pour détecter des risques d'erreurs à l'exécution ou effectuer des optimisations indépendantes du langage cible.

Par exemple détecter qu'une variable n'a pas été initialisée avant son utilisation, propager une valeur de constante connue, transformer des programmes récursifs terminaux en programme itératifs, détecter des morceaux de codes inatteignables (code-mort). Chaque analyse coûte cher mais peut permettre d'optimiser le code engendré ou bien de détecter les erreurs. Le choix des analyses dépend de ce qui est possible (beaucoup des propriétés que l'on souhaiterait établir comme le non-dépassement des bornes dans un tableau ou la terminaison des programmes ne sont pas décidables) et de ce qui est faisable, nombreuses propriétés décidables correspondent à des algorithmes coûteux.

1.3.2 Synthèse

La synthèse correspond à la phase de reconstruction de l'expression du langage cible à partir de l'arbre syntaxique.

Dans le cas de la compilation d'un langage de programmation on distinguera les phases suivantes :

Allocation mémoire

L'allocation mémoire détermine comment ordonner les entités du programme dans les mots mémoires de la machine. Elle doit tenir compte de la taille de ces mots mémoires et des *contraintes d'alignement* de la machine. Par exemple un nombre ne pourra être chargé, écrit ou utilisé dans une opération que s'il est situé sur une frontière du mot.

Le compilateur peut être amené à réserver de l'espace en mémoire pour des calculs intermédiaires ou le passage des arguments d'une procédure. Il devra rendre cet espace mémoire lorsqu'il n'est plus utile.

L'utilisateur peut souhaiter manipuler des données de taille variable, la gestion de cette espace mémoire dont l'utilisation n'est pas connue a priori peut être effectuée par l'utilisateur (on parle d'allocation *statique*) ou bien par le compilateur (on parle d'allocation *dynamique*). Celui-ci doit alors mettre en œuvre des programmes de "ramasse-miettes" (GC pour Garbage Collector ou Glaneur de Cellules) chargés de récupérer l'espace mémoire réutilisable.

Génération de code

Elle construit les instructions du programme cible. Il faut déterminer les séquences d'instructions cibles correspondant aux expressions de l'arbre syntaxique. Il faut allouer judicieusement les registres de la machine qui ont un accès plus rapide que la mémoire.

Optimisations de code cible

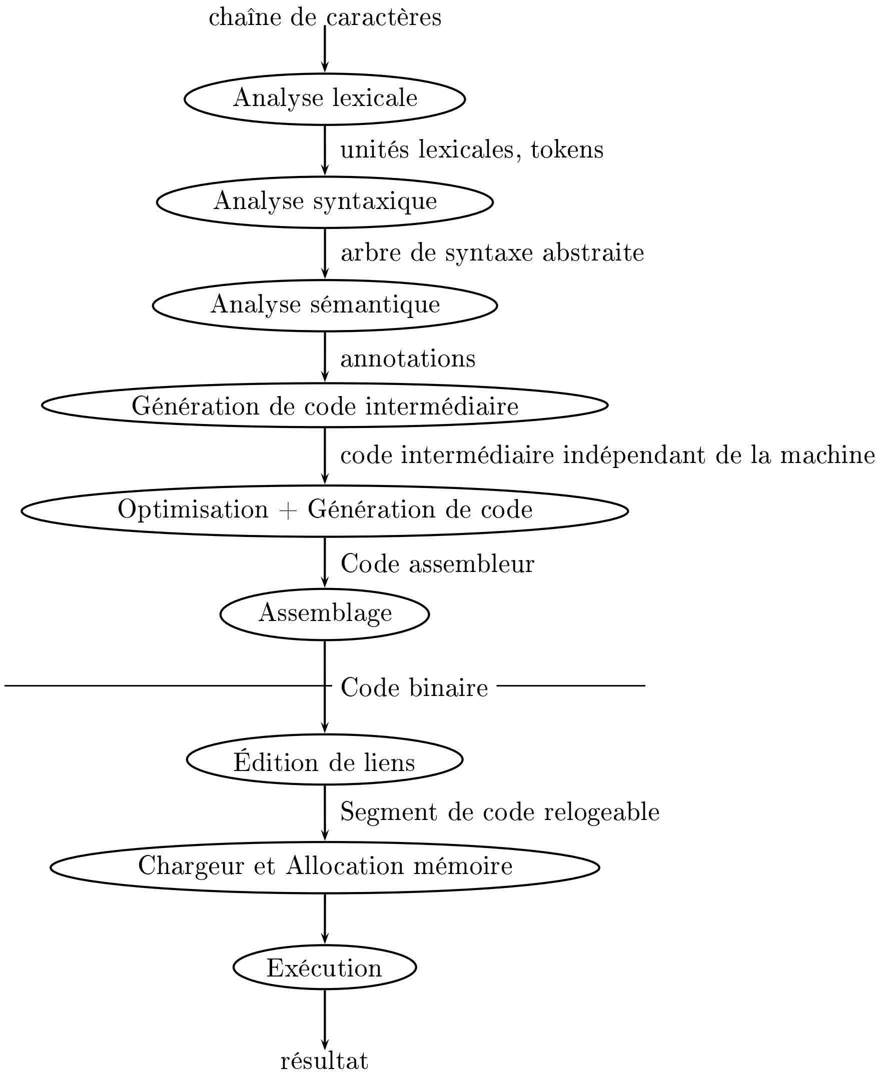
Il s'agit de détecter des séquences de code cible qui peuvent être optimisées. On peut chercher à diminuer la taille du code engendré dans le cas de processeurs aux ressources limitées.

1.3.3 Table des symboles

On construit pendant toute l'analyse une table des symboles qui associe à chaque identificateur déclaré dans le programme des informations calculées au moment de l'analyse et qui serviront au moment du traitement des occurrences des variables dans le programme. C'est par exemple le type calculé au moment de l'analyse sémantique, savoir une variable est locale, globale ou un paramètre de procédure, l'adresse mémoire utilisée au moment de l'allocation.

1.4 Architecture d'un compilateur

Le diagramme suivant décrit sommairement les différentes étapes de la compilation en distinguant la tâche du compilateur de celle du système (édition de lien, chargement et exécution).



1.4.1 Interpréteur versus Compilateur

Un *interpréteur* est un programme qui prend en entrée un programme source et les entrées de ce programme et qui calcule le résultat de l'exécution du programme sur les données.

L'interpréteur effectue en général une analyse sur la structure du programme, il ne connaît pas a priori la structure du programme qu'il aura à exécuter et ne peut donc effectuer d'optimisations.

Un compilateur effectue un pré-calcul sur un programme p pour le transformer en une suite d'instructions p'. L'exécution de p' sur les entrées doit fournir le même résultat que l'exécution de p. On s'attend à ce que le calcul du résultat soit plus efficace.

1.4.2 Machine virtuelle

C'est la définition de structures de données et d'instructions agissant sur ces données. Semblable à un langage d'assembleur, la machine virtuelle ne correspond pas à une implantation physique mais peut être vue comme un langage de bas niveau correspondant aux instructions essentielles d'une certaine classe de langage de programmation.

Cet intermédiaire permet de factoriser le travail de compilation en réduisant le travail spécifique à une architecture donnée. Il est aussi essentiel pour assurer la portabilité du code compilé (c'est ce qui est utilisé pour les applets Java par exemple).

1.4.3 Dans quel langage écrire un compilateur ?

A priori le langage dans lequel écrire un compilateur est indépendant des langages source et cible du compilateur.

On peut choisir un langage de bas niveau pour des raisons d'efficacité, ou bien un langage de haut niveau afin de mieux contrôler la complexité du développement.

Composer les compilateurs

Une compilateur C_1 de Source dans Cible écrit dans un langage L peut lui même être compilé par un compilateur de L dans M. On obtient alors toujours un compilateur de Source dans Cible mais écrit dans le langage M.

Compilateur bootstrappé

On choisit parfois d'écrire le compilateur du langage source L vers le langage machine M dans le langage L lui-même, c'est ce qu'on appelle un compilateur "bootstrappé". C'est souvent le premier gros exemple écrit dans le langage lui-même.

Pour construire le compilateur bootstrappé, il faut :

- un premier compilateur C_1 qui compile un sous-ensemble L_0 de L dans le langage machine M et est écrit en langage machine. Ce premier compilateur peut être très inefficace et ne comprend qu'une partie du langage.
- On écrit ensuite dans le langage L_0, un compilateur C_2 de L dans M qui comprend tout le langage.
- En compilant C_2 à l'aide de C_1 on a un nouveau compilateur C_3 de L dans M écrit en langage machine qui comprend tout le langage mais ne produit pas forcément du code optimisé.
- On écrit maintenant dans le langage L, un compilateur C_4 de L dans M qui comprend tout le langage et est optimisé.
- En compilant C_4 à l'aide de C_3 on a un compilateur C_5 qui produit du code optimisé mais qui est compilé à l'aide du compilateur inefficace C_3 donc qui n'est pas lui-même un programme optimisé.

- Il suffit de recompiler C_4 à l'aide de C_5 pour avoir un compilateur C_6 optimisant et optimisé.
- Recompiler C_4 à l'aide de C_6 redonne le même compilateur, il ne sert donc à rien d'essayer d'aller plus loin.

Compilation croisée

La compilation croisée (*cross-compiling* en anglais) consiste à compiler un programme source vers le langage machine M en exécutant la compilation sur une machine d'architecture différente i.e. d'un langage machine (et/ou d'un système d'exploitation) différent. Cela permet notamment de produire un programme compilé pour une machine dont on ne dispose pas (par exemple une machine d'accès restreint) ou qui n'existe pas encore (si par exemple le microprocesseur n'est pas encore construit mais qu'un émulateur est disponible pour celui-ci).

En pratique, la compilation croisée est difficile à mettre en œuvre (présence de bibliothèques, édition de liens, etc.) et peu de compilateurs la proposent.

Chapitre 2

Analyse lexicale

Ce chapitre rappelle les principaux algorithmes sur les automates finis mis en œuvre dans la reconnaissance des langages réguliers sans en donner une justification théorique complète. Il aborde également les problèmes spécifiques aux analyseurs lexicaux.

2.1 Les bases théoriques

2.1.1 Objectifs

La phase de reconnaissance syntaxique, qui déterminera si un programme est bien un élément du langage considéré, est décomposée en une phase d'analyse lexicale, une phase d'analyse syntaxique et une phase d'analyse sémantique.

L'analyse lexicale sert à reconnaître les symboles utilisés "123, toto, `begin`,..." à les classer (entiers, identificateurs, mots-clés, commentaires) en leur associant une étiquette (appelée *token*) et les traiter avant de les transmettre à l'analyseur syntaxique. On leur trouvera une représentation machine efficace par l'intermédiaire de la table des symboles.

Cette phase doit se faire rapidement on se contentera donc de reconnaître des expressions régulières (aussi appelées rationnelles) qui peuvent être identifiées par des automates à états finis.

Certains langages tels que C utilisent des pré-processeurs, un pré-processeur permet d'expanser des directives d'inclusion de bibliothèques (`#include`) ou des constantes définies par des noms symboliques (`#define`). La phase d'analyse lexicale a lieu sur le résultat du pré-processeur appliqué au programme.

2.1.2 Qu'est-ce qu'une expression régulière ?

L'ensemble des *langages réguliers* (ou rationnels) sur un alphabet A est défini de manière inductive (plus petit ensemble vérifiant des propriétés de clôtures) :

- $\emptyset$, $\{\epsilon\}$,$\{a\}, a \in A$, sont des langages réguliers,
- si L_1 et L_2 sont des langages réguliers alors il en est de même pour $L_1 \cup L_2$ et $L_1L_2 \equiv \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$,
- si L est un langage régulier alors il en est de même pour $L^* \equiv \bigcup_{n\in\mathbb{N}} \{w_1w_2 \dots w_n \mid w_i \in L\}$

L'ensemble des *expressions régulières* sur un alphabet A est défini de manière inductive par :

- $\emptyset, \epsilon$ et a pour $a \in A$ sont des expressions régulières
- Si r_1 et r_2 sont des expressions régulières alors il en est de même pour $(r_1|r_2)$, r_1r_2
- Si r est une expression régulière alors il en est de même de $(r)^*$.

Chaque expression régulière r détermine un langage régulier $L(r)$.

$$\begin{array}{ll|ll|ll} L(\emptyset) & = \emptyset & L(\epsilon) & = \{\epsilon\} & L(a) & = \{a\} \\ L((r_1|r_2)) & = L(r_1) \cup L(r_2) & L(r_1 r_2) & = L(r_1)L(r_2) & L((r)^*) & = L(r)^* \end{array}$$

On omettra les parenthèses dans les expressions régulières lorsqu'il n'y a pas ambiguïté.

2.1.3 Qu'est-ce qu'un automate ?

Un *automate fini non déterministe (AFND) sur un alphabet* A est formé d'un ensemble Q d'états dans lequel on distingue un *état initial* q_0, un sous-ensemble d'*états d'acceptation* F et d'une relation de transition Δ qui est un sous-ensemble de $Q \times A \cup \{\epsilon\} \times Q$. Si $(p, w, q) \in \Delta$, on écrira $p \xrightarrow{w} q$ où $w \in A \cup \{\epsilon\}$ est soit le mot vide soit une lettre.

L'automate est dit *déterministe (AFD)* lorsque la relation de transition est une fonction (partielle) de $Q \times A$ dans Q : à chaque état et pour chaque caractère d'entrée, il y a au plus un état vers lequel une transition peut avoir lieu.

On étend la relation $p \xrightarrow{w} q$ de manière naturelle en une relation $p \xrightarrow{w}\!\!\!\!\!\rightarrow q$ pour un mot w de longueur quelconque. La relation $p \xrightarrow{w}\!\!\!\!\!\rightarrow q$ est la plus petite relation telle que :
- $p \xrightarrow{\epsilon}\!\!\!\!\!\rightarrow p$
- si $p \xrightarrow{w} q$ alors $p \xrightarrow{w}\!\!\!\!\!\rightarrow q$
- si $p \xrightarrow{w_1}\!\!\!\!\!\rightarrow q$ et $p \xrightarrow{w_2}\!\!\!\!\!\rightarrow q$ alors $p \xrightarrow{w_1 w_2}\!\!\!\!\!\rightarrow q$

Cette relation vérifie la propriété suivante : pour $a \in A$, $p \xrightarrow{a}\!\!\!\!\!\rightarrow q$ si et seulement si il existe p' et q' tels que $p \xrightarrow{\epsilon}\!\!\!\!\!\rightarrow p'$, $p' \xrightarrow{a} q'$ et $q' \xrightarrow{\epsilon}\!\!\!\!\!\rightarrow q$.

Le *langage reconnu par un automate* (Q, q_0, F, Δ) est l'ensemble des mots w tels qu'il existe $q \in F$ tel que $q_0 \xrightarrow{w}\!\!\!\!\!\rightarrow q$. On écrira aussi $q_0 \xrightarrow{w}\!\!\!\!\!\rightarrow$.

Représentation des automates : Un automate est souvent représenté par un graphe orienté dont les sommets sont les états et les arêtes étiquetées correspondent aux transitions. On distingue les états d'acceptation (que l'on entourera d'un cercle) et l'état initial.

2.1.4 Construction de l'automate

On utilise un algorithme récursif qui prend en entrée une expression régulière r et construit un automate fini reconnaissant le langage $L(r)$. On peut de plus fixer arbitrairement deux états p et q tel que p soit l'état initial et q le seul état d'acceptation.

Exercice Que peut-on dire de l'automate lorsque $L(r) = \emptyset$?

Construction
- si $r = \emptyset$ alors l'automate a deux états distincts p et q sans transition entre (q n'est donc pas atteignable).
- si $r = \epsilon$ alors l'automate a un seul état $p = q$ et pas de transitions.
- si $r = a$ alors on met une seule transition étiquetée par a entre p et q.
- si $r = (r_1|r_2)$ on construit récursivement les automates pour r_1 entre p_1 et q_1 et r_2 entre p_2 et q_2 on ajoute des ϵ-transitions : $p \xrightarrow{\epsilon} p_1$, $p \xrightarrow{\epsilon} p_2$,$q_1 \xrightarrow{\epsilon} q$, $q_2 \xrightarrow{\epsilon} q$.

- si $r = r_1 r_2$ on choisit un nouvel état p_1 on construit récursivement les automates pour r_1 entre p et p_1 et pour r_2 entre p_1 et q et on considère l'automate obtenu en ajoutant toutes les transitions.
- si $r = (r_1)^*$ on choisit un nouvel état p_1, on construit récursivement l'automate pour r_1 entre p_1 et q on ajoute une ϵ transition entre p et q, entre q et p_1 et entre p et p_1.

Exercice Construire l'automate pour l'expression $a(a|0)*$

Remarque L'automate ainsi construit a une taille en nombre d'états et de transitions proportionnelle à la taille de l'expression régulière initiale.

2.1.5 Rendre l'automate déterministe

Il s'agit de remplacer la relation de transition par une fonction partielle qui à un état et un caractère associe au plus un nouvel état.

Pour cela l'idée est, si l'automate initial est construit sur un ensemble Q d'états, de construire un nouvel automate avec comme états des ensembles d'états de Q.

Définitions On définit l'ensemble des ϵ-successeurs d'un état p et on note ϵ–Succ(p) l'ensemble des états q tels que $p \overset{\epsilon}{\twoheadrightarrow} q$.
On note $\epsilon-\text{Succ}(P)$ pour un ensemble d'états l'union des ϵ-successeurs des éléments $p \in P$.
Attention : de tels successeurs peuvent être obtenus par une ou plusieurs transitions.

On définit l'ensemble des successeurs d'un état p pour un caractère a et on note $\text{Succ}(p, a)$ l'ensemble des états q tels que $p \xrightarrow{a} q$.

Algorithme On se donne un automate (Q, q_0, F, Δ). L'automate déterministe correspondant aura pour états des parties de Q c'est-à-dire des ensembles d'états. On notera de manière générale $\mathcal{P}(E)$ l'ensemble des parties de E et plus spécifiquement P_Q l'ensemble des parties de Q.

Un automate déterministe reconnaissant le même langage est :
- ensemble d'états : P_Q
- état initial : $\epsilon-\text{Succ}(q_0)$
- états d'acceptation : $\{q \subset Q \mid q \cap F \neq \emptyset\}$
- transitions : $\{(q, a, q') \mid q, q' \in P_Q, a \in A, \forall y \in Q. y \in q' \Leftrightarrow \exists x \in q. x \overset{a}{\twoheadrightarrow} y\}$

L'algorithme construit progressivement les états atteignables et transitions de l'automate résultat. Il est construit comme une fonction :
`determinise` : $\mathcal{P}(P_Q) \times \mathcal{P}(P_Q) \to \mathcal{P}(P_Q) \times \mathcal{P}(P_Q \times A \times P_Q)$

`determinise`(lQ, QD) prend en entrée deux ensembles d'états de P_Q, le premier correspondant aux états à traiter et le second aux états déjà traités. Le résultat est l'ensemble des états atteignables de l'automate déterministe et la relation représentant la fonction de transitions.

L'algorithme utilise le fait que :

$$x \overset{a}{\twoheadrightarrow} y \Leftrightarrow \exists x' \in \epsilon-\text{Succ}(x). \exists y'. x' \xrightarrow{a} y' \wedge y \in \epsilon-\text{Succ}(y')$$

Cette fonction utilise une fonction auxiliaire **`transq`** qui étant donné un état q et un ensemble d'actions lA trouve toutes les transitions issues de cet état par une des actions de lA et renvoie l'ensemble des nouveaux états à traiter. On note A l'ensemble de toutes les actions.

```
determinise(lQ,QD) =
  si lQ=vide alors (QD,vide)
  sinon
```

```
    choisir q dans lQ
    soit transq(q,lA) =
             si lA = vide alors (vide,vide)
             sinon choisir a dans lA
               soit (qD,delta) = transq(q,lA\a)
               dans soit q'=epsilon-succ(succ(q,a))
               dans si q'= vide alors (qD,delta)
                    sinon soit qD'= si q' appartient à QD
                                    alors qD sinon add(q',qD)
                    dans (qD',add ((q,a,q'),delta))
     dans soit (qD,delta) = transq(q,A)
          dans soit (QD',Delta') = determinise(lQ\q union qD,add q QD)
               dans (QD',delta union Delta')
```

`determinise`$(\epsilon-\mathrm{Succ}(q_0), \emptyset)$ renvoie les états et les transitions d'un automate déterminisé reconnaissant le même langage que l'automate initial.

Construction directe d'automates déterministes à partir d'expression régulières Il est également possible de construire directement un automate déterministe à partir d'une expression régulière (cf exercice de TD).

2.1.6 La minimalisation

Les automates obtenus peuvent être très gros (l'ensemble des parties d'un ensemble à n éléments est de cardinal 2^n) et posséder des états distincts à partir desquels on reconnaît les mêmes langages.

On va regrouper de tels états. Pour cela on va partitionner l'ensemble des états pour regrouper ceux qui sont indistingables vis-à-vis du langage reconnu.

On dit que $p \sim q$ si pour tout mot w, $p \xrightarrow{w}$ si et seulement si $q \xrightarrow{w}$.

Cette propriété peut paraître non constructible en fait on introduit une relation plus grossière : $p \sim_n q$ si pour tout mot w de longueur au plus n, $p \xrightarrow{w}$ si et seulement si $q \xrightarrow{w}$.

On construit les classes d'équivalence de $\sim_{n+1}$ en fonction de celles de $\sim_n$. Lorsqu'un point fixe est obtenu alors on a la partition souhaitée.

Algorithme : L'algorithme sépare progressivement les ensembles d'états de l'automate donné en entrée qui se distinguent par la reconnaissance de mots différents.

Il est construit autour d'une fonction :
`raffine` : $\mathcal{P}(P_Q) \to \mathcal{P}(P_Q)$ qui prend en entrée une partition de l'ensemble des états correspondant à la relation $\sim_n$, les éléments de cette partition à traiter et renvoie la partition correspondant à $\sim_{n+1}$.

$$\texttt{raffine}(Q_n) = \{\{q \in qn \mid \mathrm{Succ}(q,a) \in qn'\} \neq \emptyset \mid qn, qn' \in Q_n, a \in A\}$$

La fonction `minimise`, pour un automate (Q, q_0, F, Δ) part d'une partition pour $\sim_0$ qui n'est autre que $\{F, Q\backslash F\}$ et itère la fonction `raffine` jusqu'à ce que le résultat de `raffine`(Q_n) soit égal à Q_n.

2.1.7 Exercice : Recherche de motifs

Soit R une expression régulière reconnue par un automate (Q, q, F, Δ), on ajoute à cet automate deux nouveaux états q_0 et f avec des ϵ-transitions de q_0 vers q et de F vers f et pour chaque $a \in A$ une a-transition de q_0 dans lui-même et de f sur lui-même.

- Quel langage est reconnu par cet automate ?
- Déterminiser l'automate
- Quelle est la complexité de l'algorithme de reconnaissance obtenu ?

2.1.8 Reconnaissance des commentaires

La reconnaissance des commentaires compris entre R_1 et R_2 correspond à une expression régulière $R_1(\overline{A^*R_2A^*})R_2$ qui s'interprète comme : lire R_1, lire une expression ne comportant pas R_2 puis lire R_2. Pour cela on peut effectuer la séquence suivante :

- Construire un automate pour R_2
- Construire un automate pour $A^*R_2A^*$
- Déterminiser cet automate
- Inverser l'automate déterminisé
- Construire l'automate en ajoutant la reconnaissance de R_1 au début et R_2 à la fin.
- Le rendre déterministe.

Certains langages utilisent des commentaires imbriqués. Ceci permet de pouvoir mettre en commentaire une partie de code même si celle-ci comporte déjà des commentaires. Une telle analyse sort du cadre de ce qui est reconnaissable par un automate, néanmoins les générateurs d'analyseurs lexicaux offrent des constructions permettant de gérer de telles situations.

2.1.9 A propos de l'efficacité de la reconnaissance

Un automate fini peut être stocké en utilisant une place proportionnelle à son nombre de transitions.

Les automates déterministes peuvent avoir un nombre d'états exponentiel par rapport à la taille d'un automate non-déterministe reconnaissant le même langage. C'est le cas de la reconnaissance du langage contenant tous les mots construits sur l'alphabet {a,b} qui ont un a exactement n lettres avant la fin. Ce langage correspond à l'expression régulière

$$(a|b) * a \underbrace{(a|b) \ldots (a|b)}_{n-1 \text{ fois}}$$

Ce langage peut être reconnu par un AFND dont le nombre d'états est proportionnel à n mais le nombre d'états pour un AFD reconnaissant le même langage est proportionnel à 2^n, chaque état devant représenter les positions des a rencontrés dans les n derniers caractères.

Reconnaissance à partir d'un automate non-déterministe On peut faire la reconnaissance à partir d'un automate non déterministe en simulant l'automate déterministe correspondant. Pour cela on utilise deux piles permettant de stocker l'ensemble des états non-déterministes représentant l'état courant, et l'ensemble des états non-déterministes représentant l'état résultat de la transition sur le caractère a reconnu. On dispose de plus d'un tableau permettant de savoir si un état est déjà dans la pile résultat. Pour chaque état de la pile initiale, on calcule les états obtenus par transition sur a puis on effectue une clôture par les ϵ-transitions. Le calcul de chaque transition est proportionnel au nombre de transitions dans l'automate non déterministe, qui est lui-même, avec la construction proposée, proportionnel à la taille de l'expression régulière. La reconnaissance d'un mot de longueur n dans un langage correspondant à une expression régulière de taille p nécessite un temps proportionnel à $n \times p$.

Si on ne peut stocker la table de transitions complète pour un automate déterministe reconnaissant le langage, on peut utiliser des méthodes paresseuses. La table de transition est construite en fonction des besoins, à chaque transition on vérifie si celle-ci n'a pas déjà été calculée, sinon on la recalcule. Lorsque la table devient trop grosse, il est possible d'éliminer les transitions les moins utiles qui seront recalculées si nécessaires.

2.2 Construction d'analyseurs lexicaux

Pour construire un analyseur lexical on décrit d'abord informellement puis par une expression régulière les classes de symboles à identifier. On construit ensuite un automate fini déterministe (si possible minimal) reconnaissant le langage qui est l'union des langages correspondant à chaque classe de symboles.

2.2.1 Fonctionnement de l'analyseur

Soit un automate et un mot d'entrée. On cherche à savoir si le mot appartient au langage correspondant à l'automate. L'état de l'analyseur est caractérisé par : l'état courant q de l'automate, le mot reconnu jusqu'à q, le mot restant à analyser. Tant que le mot restant à analyser n'est pas vide, l'analyseur lit un caractère d'entrée et détermine si une transition est possible sur ce caractère à partir de l'état courant. Si oui l'état courant devient l'état résultant de la transition sinon il y a une erreur, le mot ne fait pas partie du langage. Le mot est reconnu lorsqu'après lecture, on se trouve dans un état d'acceptation.

Dans le cas d'un analyseur lexical, les états d'acceptation sont de plus décorés de manière à distinguer les différentes classes d'unités lexicales reconnues. Il ne s'agit pas seulement de reconnaître que le mot d'entrée est un élément du langage mais de le découper en une suite d'unités lexicales.

Comme une unité lexicale peut avoir un préfixe qui est également une unité lexicale il peut y avoir ambiguïté.

La convention est de chercher à reconnaître l'unité lexicale la plus longue. Par exemple si `>` et `>=` sont deux unités lexicales on voudra ne reconnaître `>` comme une unité que s'il n'est pas suivi d'un `=`. De même un identificateur peut contenir un préfixe qui est un mot clé, il faut alors reconnaître l'identificateur et non le préfixe.

L'état de l'analyseur lexical nécessite donc une information supplémentaire correspondant à la position dans le texte analysé du dernier état final rencontré. Lorsque l'analyseur passe par un état final, il en mémorise la position et continue la recherche d'une unité plus longue. Lorsqu'il n'y a plus de transition applicable deux situations peuvent se présenter :
- Aucune position finale n'a été mémorisée, c'est un cas d'échec de l'analyse lexicale.
- Le mot reconnu jusque là est wv avec w l'unité lexicale reconnue au dernier passage par un état final. Soit l le mot restant à reconnaître, l'analyse redémarre à l'état initial avec l'entrée vl.

2.2.2 Outils d'analyse lexicale

L'écriture d'un analyseur lexical à la main est possible mais souvent moins efficace que l'utilisation d'un générateur d'analyseur lexicaux.

Les outils d'analyse lexicale tels que lex, jflex, ocamllex, etc., prennent en entrée un fichier source contenant les expressions régulières et les actions associées et après compilation produisent un fichier du langage cible associé (C dans le cas de lex, Java dans le cas de jflex, Ocaml dans le cas de ocamllex, etc.). Le fichier cible obtenu doit alors être compilé en utilisant un compilateur pour le code cible.

2.2.3 Expressions régulières étendues

Écrire une expression régulière correspondant par exemple aux flottants est très lourd ; il faut donc des notations de plus haut niveau.

Classes Il est utile de regrouper des ensembles de caractères qui jouent le même rôle dans une expression régulière.

Par exemple "`a-z`" représente l'ensemble des caractères compris entre les caractères `a` et `z` dans l'ordre des caractères ASCII. De même "`0-9`" représente l'ensemble des chiffres.

Introduire une classe permet de mettre ensemble des caractères qui ont le même statut pour diminuer le nombre de transitions par exemple :

digit = [0-9]
char = [a-z | A-Z]

Les classes doivent être disjointes, si elles ne le sont pas il faut les partitionner de manière qu'à chaque caractère corresponde une classe. L'automate ne s'occupera que des classes.

Notations Certaines conventions permettent de désigner de manière concise des expressions régulières composées.

Par exemple si r désigne une expression régulière :

[^s]	avec s désignant un ensemble de caractères, reconnaît les caractères autres que ceux dans s
"w"	reconnaît la chaîne de caractères w
r ?	équivalent à $(r \mid \epsilon)$, une au plus occurrence de mots dans $L(r)$
r +	équivalent à rr^*, une ou plusieurs occurrences de mots dans $L(r)$
r{n}	avec n un entier, désigne les mots formés de n occurrences de mots dans $L(r)$
r{n,m}	avec n et m des entiers, désigne les mots formés de k occurrences de mots dans $L(r)$ avec $n \leq k \leq m$

Les différents générateurs d'analyseur lexicaux utilisent des conventions particulières pour désigner certains ensembles de caractères.

Abréviations Il peut être utile d'introduire également des noms servant d'abréviation pour des expressions régulières.

Exemple

digit = 0-9
char = a-z|A-Z
int=digit digit*
float = int.int (E digit digit |ϵ)

Les noms servent également à éviter des conflits entre le langage analysé et les signes représentant des expressions régulières.

2.2.4 Génération de l'analyseur

On se donne une suite de définitions $A_i = R_i$, avec A_i un nouveau nom et R_i une expression régulière pouvant faire intervenir des noms A_j pour $j < i$. On substitue de manière à obtenir des expressions régulières où A_j n'apparaît plus dans R_i.

On construit pour chaque expression un AFND sur un ensemble d'états qui lui est propre. Les états d'acceptation sont étiquetés par un *token* qui identifie la classe d'unités lexicales reconnue. On fait la réunion de tous ces automates en ajoutant un état initial qui va avec une ϵ-transition vers chaque état initial. On peut ensuite rendre cet automate déterministe. Lors de la déterminisation de l'automate, il se peut que plusieurs états d'acceptation correspondant à différents token se retrouvent dans le même état. Cela correspond à une situation où la spécification des classes d'unités lexicales est ambiguë. Le concepteur de l'analyseur doit faire un choix sur le token à renvoyer. On peut éventuellement minimiser cet automate, en adaptant l'algorithme vu précédemment en séparant les états d'acceptation correspondant à des classes d'unités distinctes.

2.2.5 Représentation de la table de transitions

On peut représenter un automate non déterministe par une matrice dont les lignes correspondent aux états et les colonnes aux caractères et au mot vide. Chaque case (q, w) contient l'ensemble des états atteignables à partir de l'état q en effectuant une transition étiquetée par w.

Dans le cas des automates déterministes, les colonnes correspondent uniquement aux caractères (ou aux classes de caractères) et chaque entrée de la matrice contient au plus un état. Ces matrices sont essentiellement creuses ce qui permet d'en optimiser la représentation.

Les caractères d'entrée sont regroupés en classes disjointes numérotées de 1 à n. On utilise un tableau dont les lignes sont les états et les colonnes les entiers correspondant aux classes de caractères. Le tableau contient le prochain état à la lecture d'un caractère de la classe. Ce tableau est gros mais contient en général beaucoup de cases identiques, (par exemple correspondant à l'absence de transition). On peut le compacter en supprimant la transition la plus courante pour un état s qui est stockée dans un tableau `defaut`. On peut ensuite ranger les peignes correspondant aux autres transitions dans un seul tableau linéaire `trans`, un autre tableau `verifie` de même taille indique à quel état est associé chaque transition.

On garde un tableau `index` indexé par les états. La ligne correspondant à un état q est stockée dans `trans` à partir de la valeur $\texttt{index}(q)$. Ainsi l'état résultat de la lecture d'un caractère dans la classe n à partir de l'état q se lit à l'indice $p = (\texttt{index}(q) + n)$. Si $\texttt{verifie}(p) = q$ alors l'état résultat de la transition est bien $\texttt{trans}(p)$ sinon il s'agit de $\texttt{defaut}(q)$. On pourrait aussi au lieu de choisir pour $\texttt{defaut}(q)$ un état par défaut, choisir un état de référence, dans les situations par défaut, le résultat de la transition se lira à l'adresse $(\texttt{index}(\texttt{defaut}(q)) + n)$.

2.2.6 Traitement de l'entrée

L'entrée de l'analyseur lexical se fait caractère par caractère à partir d'un fichier. L'accès au fichier étant une opération coûteuse, on utilisera un tableau (buffer) pour lire une suite de caractères. L'analyseur lexical traitant directement les entrées à partir de ce tableau. La taille d'une unité lexicale ne doit pas dépasser la taille de ce tableau (ce qui impose une limite au chaînes de caractères ou aux identificateurs). Cependant il peut arriver qu'une unité lexicale soit coupée entre deux portions obtenues par accès au disque. Pour gérer efficacement ce problème, on choisit de doubler la taille du tableau qui est séparé en deux. Chaque fois que l'on sort de l'une des demi-portions, on effectue une lecture sur l'autre demi-portion. Trois pointeurs repèrent dans le tableau le début de l'unité lexicale, le caractère courant, la fin de la plus grande unité lexicale reconnue. Lorsque l'unité lexicale à renvoyer est reconnue, deux cas de figure peuvent se présenter :

- Le pointeur de fin d'unité lexicale est après celui de début d'unité lexicale et la valeur de l'unité lexicale est la chaîne de caractères située entre les deux.
- Le pointeur de fin d'unité lexicale est avant celui de début d'unité lexicale et la valeur de l'unité lexicale est la chaîne obtenue par concaténation de la chaîne entre le pointeur de début de l'unité lexicale et la fin du tableau et de la chaîne entre le début du tableau et le pointeur de fin de l'unité lexicale.

Remarque Certains langages ont besoin pour la reconnaissance d'une unité lexicale de connaître une partie des caractères suivants de l'entrée. C'est le cas de langages tels que FORTRAN pour lesquels les blancs ne sont pas significatifs et certains mots-clé ne sont distingués des identificateurs que par leur contexte d'utilisation. Il y a confusion entre la construction : $\mathtt{IF}(I_1, .., I_n) = expr$ qui représente l'affectation à un tableau représenté par l'identificateur `IF` et les conditionnelles dans lesquelles `IF` joue le rôle de mot-clé et qui s'écrivent : `IF` (*cond*) `THEN` *bloc* ou bien : `IF` (*cond*) `THEN` *bloc* `ELSE` *bloc*

Remarque Les langages "modernes" évitent ce genre de complication en donnant aux mot-clés le statut de mots réservés qui ne peuvent être utilisés comme identificateurs.

Si r_1 et r_2 sont deux expressions régulières, un analyseur comme lex gère l'expression r_1/r_2 qui signifie : reconnaître une unité lexicale appartenant au langage $L(r_1)$ si elle est suivie d'une expression du langage $L(r_2)$. En pratique il suffit de construire l'automate pour r_1r_2, en mémorisant comme fin de l'unité lexicale l'état d'acceptation de r_1, cette unité n'étant valide que si la reconnaissance permet d'atteindre l'état d'acceptation de r_1r_2. On remarquera que cette construction nécessite que le buffer d'entrée dépasse la taille de la plus grande unité lexicale et soit assez grand pour contenir les mots du langage r_1r_2.

2.2.7 Interprétation vs compilation de l'analyseur

L'analyseur engendré à partir de la description des expressions régulières peut être interprété : un analyseur générique est écrit qui utilise une table de transition calculée automatiquement. Un générateur tel que flex est plus efficace, il engendre directement un programme où les différentes possibilités à la lecture d'un caractère d'entrée sont compilées sous forme d'une analyse par cas sur le caractère d'entrée.

2.3 Traitement des unités lexicales

2.3.1 Le rôle du crible

Le crible sert à faire des traitements sur le résultat de l'analyseur lexical. Ces traitements peuvent être :

- Parmi la classe des identificateurs reconnaître les mots clés (il peut être plus judicieux de séparer cette phase plutôt que de surcharger l'analyseur lexical)
- Ranger les valeurs infinies reconnues dans une table en leur associant un numéro. La stratégie sera différente suivant si on cherche à distinguer ou non des valeurs différentes. Dans le cas des identificateurs, on souhaite reconnaître au plus tôt deux objets égaux, ce n'est pas forcément le cas des chaînes de caractères.
- Ne pas renvoyer à l'analyseur syntaxique certaines classes de symboles (commentaires, espaces).

2.3.2 Actions

Dans les analyseurs lexicaux tels que lex, le crible sera traité dans les actions associées à chaque règle. Chaque action est associée à une expression régulière et déclenchée lorsque l'unité lexicale reconnue appartient au langage associé à cette expression (dans le cas où l'unité appartient à plusieurs expressions, c'est la première dans le fichier de description qui est choisie). L'action fait référence à des informations issues de l'analyse lexicale telles que la longueur de l'unité lexicale ou la chaîne de caractères associée.

Les actions associées à chaque règle permettent d'étendre la classe des langages reconnus par les analyseurs lexicaux au delà des langages réguliers.

2.3.3 Liaison entre l'analyseur lexical et l'analyseur syntaxique

L'analyseur lexical va fournir au fur et à mesure de la reconnaissance, des unités lexicales à l'analyseur syntaxique.

Chaque unité lexicale est composée de deux informations :

- le *token* qui représente la classe à laquelle appartient l'unité lexicale (identificateur, entier, réel,...) . Ce token est symbolisé par un nom choisi librement mais pourra être représenté de manière interne plus efficacement par un chiffre, le nom étant souvent gardé pour des raisons d'interface avec l'utilisateur.

  L'ensemble des tokens forme l'alphabet des terminaux de la grammaire.

  Dans jflex, une instruction du type `{return new Symbol(sym.CLASSE) ;}` apparaîtra à la fin de chaque action correspondant à la reconnaissance d'une unité de la classe correspondante (où la classe `sym` regroupe les constantes correspondant aux différentes classes ; voir le chapitre suivant).
- lorsque la classe correspond à plusieurs unités lexicales, la *valeur de l'unité lexicale* aussi appelée *léxème* qui peut être une valeur entière, réelle, un pointeur sur la table des symboles, etc.

  Dans jflex, une instruction du type `{return new Symbol(sym.CLASSE, <valeur>) ;}` apparaîtra dans chaque action correspondant à la reconnaissance d'une unité de la classe correspondante. La valeur sera alors accessible dans les actions sémantiques de la grammaire. Elle est en revanche inutile pour la phase simple de reconnaissance syntaxique du langage.

Chapitre 3

Analyse syntaxique

Ce chapitre rappelle les principales définitions et algorithmes concernant les grammaires. Il présente les principes de fonctionnement de l'analyse descendante et de l'analyse ascendante. Il s'intéresse finalement à calculer pour des classes de grammaires bien identifiées les tables de transitions pour ces deux types d'analyse.

3.1 Généralités

3.1.1 Objectifs

L'analyseur syntaxique prend comme entrée une suite de tokens qui lui est fournie par l'analyseur lexical. Il reconnaît si l'entrée appartient au langage étudié. Les attributs de la grammaire permettront de traiter la valeur sémantique du programme et de rejeter certains programmes.

3.1.2 Arbre de syntaxe abstraite

La valeur sémantique d'un programme peut souvent être utilement représentée par un *arbre de syntaxe abstraite* qui donne une vision structurée du programme facilitant les traitements ultérieurs tels que la vérification du typage, les transformations de programme ou bien la génération de code.

Pour un programme, on pourra identifier les déclarations de variables, les procédures ainsi que le corps du programme. À l'intérieur du programme on reconnaîtra les instructions élémentaires et les différentes structures de contrôle.

Exemple Une structure de boucle peut être décrite dans la syntaxe concrète par l'une des formes suivantes :

```
while b do begin i1; i2 end
while b do i1; i2 done
tant que b faire i1 puis i2
```

pour les traitements ultérieurs on ne souhaite retenir que le fait qu'on a une boucle *tant-que* dont la condition est b et dont le corps est la liste d'instructions `i1;i2`.

Cette forme doit être adaptée aux transformations futures que devra subir le programme.

Certaines constructions sont appelées *sucre syntaxique* il s'agit de constructions syntaxiques pouvant être définies en termes d'autres constructions du langage considérées comme plus primitives.

Ainsi l'instruction `repeat i until b` pourrait être considérée comme une simple abréviation pour la construction `i ; while not b do i`. Dans ce cas il n'est pas utile d'avoir deux constructions distinctes dans l'arbre de syntaxe abstraite.

Remarque Les compilateurs anciens évitaient de construire l'arbre de syntaxe abstraite qui était coûteux en place. Cependant faire l'analyse sémantique et la génération de code directement dans les attributs de la grammaire n'est pas un facteur de lisibilité ni de modularité.

3.1.3 Traitement de l'analyse syntaxique

L'analyse syntaxique s'attache à vérifier que l'entrée est bien un élément d'un langage. Dans le cas des langages de programmation, pour des raisons d'efficacité, on s'intéresse à des langages décrits par des *grammaires non contextuelles* qui peuvent être reconnues par des *automates à piles*.

Traitement des erreurs Une tache importante de l'analyseur syntaxique est de détecter les erreurs. En effet un compilateur en général a à traiter des programmes syntaxiquement erronés. On peut lui demander plusieurs choses :
- localiser et annoncer l'erreur
- émettre un diagnostic
- corriger l'erreur
- reprendre l'analyse pour découvrir de nouvelles erreurs (pour des fonctionnements non interactifs)

Certaines grammaires ont une propriété de préfixe qui dit que si l'analyseur a pu traiter un préfixe u d'un mot uv alors u est correct dans le sens où il existe w tel que uw soit dans le langage reconnu par la grammaire. De telles grammaires permettront de localiser les erreurs plus précisément.

3.2 Grammaires

3.2.1 Définitions

Une *grammaire* est définie par deux alphabets disjoints N (les non-terminaux) et T (les terminaux). On distingue un non-terminal S appelé *symbole de départ*. On se donne un ensemble P de *productions* qui est un sous-ensemble de $N \times (N \cup T)^*$. On notera $X ::= \omega$ une production avec $X \in N$ et ω un mot formé sur l'alphabet $(N \cup T)^*$.

Un mot α *se dérive en une étape* en un mot β par une grammaire (N, T, S, P) et on note $\alpha \Rightarrow \beta$ si il existe $\alpha_1, \alpha_2 \in (N \cup T)^*$ et $X \in N$ tels que :

$$\begin{array}{l} \alpha = \alpha_1 X \alpha_2 \\ \beta = \alpha_1 \omega \alpha_2 \\ X ::= \omega \in P \end{array}$$

La dérivation est dite *gauche* si de plus $\alpha_1 \in T^*$ (on réécrit le non-terminal le plus à gauche) et droite si $\alpha_2 \in T^*$ (on réécrit le non-terminal le plus à droite).

On notera $\stackrel{*}{\Rightarrow}$ la fermeture réflexive-transitive de $\Rightarrow$ c'est-à-dire la plus petite relation réflexive transitive contenant $\Rightarrow$.

On dit que $m \in T^*$ est reconnu par une grammaire G si $S \stackrel{*}{\Rightarrow} m$.

Le *langage reconnu par une grammaire* est l'ensemble des mots reconnus.

On s'intéresse dans la suite à des grammaires *réduites* c'est-à-dire telles tous les non-terminaux A sont *productifs* i.e. $\exists m \in T^*. A \stackrel{*}{\Rightarrow} m$ et *atteignables* i.e. $\exists \alpha, \beta \in (N \cup T)^*. S \stackrel{*}{\Rightarrow} \alpha A \beta$.

Propriété On se sert souvent dans les démonstrations de la propriété suivante qui traduit le fait que la grammaire est non contextuelle.

Si α_1, α_2 et β sont des mots de $(N \cup T)^*$ et si $\alpha_1\alpha_2 \overset{*}{\Rightarrow} \beta$ alors il existe $\beta_1, \beta_2 \in (N \cup T)^*$ tel que

$$\alpha_1 \overset{*}{\Rightarrow} \beta_1,\ \alpha_2 \overset{*}{\Rightarrow} \beta_2,\ \text{et } \beta = \beta_1\beta_2$$

les dérivations de $\alpha_1 \overset{*}{\Rightarrow} \beta_1$ et $\alpha_2 \overset{*}{\Rightarrow} \beta_2$ sont de longueur inférieure ou égale à la dérivation de $\alpha_1\alpha_2 \overset{*}{\Rightarrow} \beta$.

Notations On factorise parfois plusieurs productions ayant le même non-terminal gauche en écrivant :

$$X ::= w_1 \mid w_2 \mid w_3$$

pour représenter les trois productions $X ::= w_1$, $X ::= w_2$ et $X ::= w_3$. On évitera de confondre la notation $\mid$ avec l'un des terminaux du langage.

3.2.2 Arbre de dérivation syntaxique

Un *arbre de dérivation syntaxique* pour la grammaire G de racine $X \in N$ et de feuilles $\omega \in T^*$ est un arbre ordonné dont la racine est X, les feuilles sont étiquetées par des terminaux formant le mot ω et les nœuds internes par des non-terminaux tels que si Y est un nœud interne dont les p fils sont étiquetés par les symboles $a_1 \ldots a_p$ alors $Y ::= a_1 \ldots a_p$ est une production de P.

Les arbres peuvent se construire à partir des dérivations et réciproquement. Plusieurs dérivations peuvent donner le même arbre. Pour établir une propriété des mots reconnus par une grammaire, plutôt que de raisonner par récurrence sur la longueur d'une dérivation, il est souvent plus commode d'utiliser une *récurrence structurelle* sur l'arbre de dérivation syntaxique.

Une grammaire est *non-ambiguë* si tout mot est reconnu par au plus un arbre syntaxique. De manière équivalente, si tout mot admet une seule dérivation gauche (resp. droite).

La grammaire des expressions suivantes est ambiguë, il y a deux arbres syntaxiques possibles pour $x + x * x$

$$\begin{array}{l} E ::= E + E \\ E ::= E * E \\ E ::= x \end{array}$$

Remarque Il ne faut pas confondre l'arbre de dérivation syntaxique qui traduit la reconnaissance du programme par la grammaire et l'arbre de syntaxe abstraite qui lui reflète la structure sémantique du programme et doit être adapté aux traitements ultérieurs.

3.2.3 Quelques exemples

Le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ (typiquement un langage de parenthèses) n'est pas régulier mais est reconnaissable par une grammaire non contextuelle.

$$\begin{array}{l} X ::= aXb \\ X ::= \epsilon \end{array}$$

Le langage $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable par une grammaire non contextuelle.

3.2.4 Grammaire et décidabilité

Il n'est pas décidable de savoir si :
- une grammaire non contextuelle est ambiguë
- deux grammaires non contextuelles reconnaissent le même langage
- l'intersection de deux grammaires non contextuelles est vide

3.2.5 Analyse descendante/ascendante

Analyser un mot, c'est établir si le mot appartient au langage engendré par la grammaire. En pratique on construit également dans le cas positif une dérivation du mot à partir du symbole initial de la grammaire ou bien un arbre de dérivation syntaxique.

Soit un mot à reconnaître, l'analyse descendante part du symbole de départ et l'expanse jusqu'à obtenir le mot, l'arbre de dérivation syntaxique est donc construit par le haut. L'analyse ascendante cherche au contraire à factoriser le mot en reconnaissant des parties droites de production jusqu'à retomber sur le symbole de départ, l'arbre de dérivation syntaxique est donc construit à partir de ses feuilles en regroupant des forêts.

3.3 Automates à pile

Tout comme les langages réguliers sont reconnus par des automates finis, les langages non contextuels sont reconnus par des automates à pile. La pile permet de mémoriser des informations au cours des transformations pour décider de la prochaine transition à effectuer, cette pile étant non bornée peut mémoriser des informations de taille variable en fonction de l'entrée.

3.3.1 Définitions

Un *automate à pile* sur un alphabet A est défini par un ensemble d'états Q, un état initial $q_0 \in Q$, un ensemble d'états d'acceptation $F \subset Q$ et une fonction de transition $\Delta \in Q^+ \times A \cup \{\epsilon\} \times Q^*$.

Remarque Contrairement à l'automate fini, la fonction de transition de l'automate à pile décrit la transformation du sommet de la *pile* qui est une suite finie d'états représentée comme un mot de Q^+.

La fonction de transition de l'automate définit plus largement une fonction de transformation de la pile : la transition est appliquée au sommet de la pile.

Si $m_1 \xrightarrow{x} m_2$ avec $x \in A \cup \{\epsilon\}$ alors $mm_1 \overset{x}{\twoheadrightarrow} mm_2$. Comme pour les automates finis, cette relation est étendue à un mot quelconque : si $m_1 \overset{w_1}{\twoheadrightarrow} m_2$ et $m_2 \overset{w_2}{\twoheadrightarrow} m_3$ alors $m_1 \overset{w_1 w_2}{\twoheadrightarrow} m_3$.

Un mot w est *reconnu par l'automate à pile* si $q_0 \overset{w}{\twoheadrightarrow} mf$ avec $f \in F$ un état final qui se retrouve au sommet de la pile.

Un automate à pile est *déterministe* si et seulement si pour tous $m_1, m_2, m_1', m_2' \in Q$ et $a, a' \in A \cup \{\epsilon\}$ si $m_1 \xrightarrow{a} m_2$ et $m_1' \xrightarrow{a'} m_2' \in \Delta$ et m_1 est un suffixe de m_1' alors $m_1 = m_1', m_2 = m_2'$ et $a = a'$.

3.3.2 Construction d'un automate à pile

On se donne une grammaire non contextuelle (N, T, S, P). On définit les *items* (non-contextuels) de G qui sont des triplets $(Y, \alpha, \beta) \in N \times (N \cup T)^* \times (N \cup T)^*$ $Y ::= \alpha\beta \in P$. On notera un tel triplet

$$[Y \to \alpha.\beta]$$

Si $\alpha = \epsilon$ alors on écrit $[Y \to .\beta]$ si $\beta = \epsilon$ alors on écrit $[Y \to \alpha.]$ et l'item est dit complet, si $\alpha = \beta = \epsilon$ on écrit simplement $[Y \to .]$

Pour construire l'automate à pile correspondant à la grammaire, on introduit un nouveau symbole de départ S'.

L'automate à pile est construit avec pour alphabet, l'ensemble T des terminaux et pour états l'ensemble des items.

L'état initial est $S' \to .S$: on cherche à reconnaître S à partir de S'. Le seul état d'acceptation est $S' \to S.$ qui signifie qu'un mot dérivable à partir de S a été reconnu.

Les transitions sont de trois sortes :

- Expansion : pour chaque $Y ::= \alpha \in P$

$$[X \to \beta.Y\gamma] \xrightarrow{\epsilon} [X \to \beta.Y\gamma][Y \to .\alpha]$$

- Lecture $a \in T$

$$[X \to \beta.a\gamma] \xrightarrow{a} [X \to \beta a.\gamma]$$

- Réduction

$$[X \to \beta.Y\gamma][Y \to \alpha.] \xrightarrow{\epsilon} [X \to \beta Y.\gamma]$$

Exemple La reconnaissance sur a^2b^2 se passe ainsi :

	$[S' \to .S]$			
expansion $S ::= aSb$	$[S' \to .S]$	$[S \to .aSb]$		
lecture a	$[S \to a.Sb]$			
expansion $S ::= aSb$	$[S' \to .S]$	$[S \to a.Sb]$	$[S \to .aSb]$	
lecture a	$[S' \to .S]$	$[S \to a.Sb]$	$[S \to a.Sb]$	
expansion $S ::= \epsilon$	$[S' \to .S]$	$[S \to a.Sb]$	$[S \to a.Sb]$	$[S \to .]$
reduction	$[S' \to .S]$	$[S \to a.Sb]$	$[S \to aS.b]$	
lecture b	$[S' \to .S]$	$[S \to a.Sb]$	$[S \to aSb.]$	
reduction	$[S' \to .S]$	$[S \to aS.b]$		
lecture b	$[S' \to .S]$	$[S \to aSb.]$		
reduction	$[S' \to S.]$			

Un invariant est conservé qui dit que si à partir de l'état initial on a reconnu le mot u et que l'état de la pile est

$$[X_1 \to \alpha_1.\beta_1] \ldots [X_n \to \alpha_n.\beta_n]$$

alors il y a une dérivation dans la grammaire G de $\alpha_1 \ldots \alpha_n$ vers u.

Cet automate n'est pas déterministe du fait des choix à faire pour les expansions.

Dans une exécution de cet automate à pile on peut :

- soit garder trace des transitions d'expansion et on pourra reconstruire une dérivation gauche.
- soit garder trace des transitions de réduction et on pourra reconstruire une dérivation droite.

Exemple Analyse de $id + id * id$ avec la grammaire des expressions arithmétiques :

$$\begin{array}{ll} E & ::= T \mid E\ + T \\ T & ::= F \mid T\ * F \\ F & ::= id \mid (\ E\ ) \end{array}$$

3.4 Analyse descendante

Le problème se pose de comment choisir une règle d'expansion pour un non terminal. Une idée naturelle est de regarder un ou plusieurs caractères de l'entrée pour choisir la bonne dérivation.

3.4.1 Introduction

En général un non-terminal Y apparaît dans plusieurs règles $Y ::= \alpha_1, \ldots, Y ::= \alpha_n$. Si chaque α_i commence par un terminal différent alors on peut décider de la dérivation en regardant un caractère. De manière générale il est possible de calculer tous les premiers terminaux possibles des mots issus d'une dérivation $Y ::= \alpha_i$. Si les ensembles sont disjoints alors la lecture du non-terminal détermine de manière unique la dérivation à appliquer.

Exemple

$$\begin{array}{lll} 1 & S & := AdS \\ 2 & S & := b \\ 3 & A & := aAb \\ 4 & A & := c \end{array}$$

Pour réécrire le non-terminal S on choisira la production 1 si l'entrée est a ou c et la production 2 si l'entrée est b.
Pour réécrire le non-terminal A on choisira la production 3 si l'entrée est a et la production 4 si l'entrée est c.

Il y a une difficulté lorsque le mot vide appartient au langage reconnu à partir d'un non-terminal. Si par exemple on ajoute à la grammaire précédente la production :

$$5 \quad A \quad := \epsilon$$

ϵ appartient au langage reconnu à partir du non-terminal A, pour décider si la production 1 doit être utilisée il ne suffit pas de regarder le premier caractère de A mais il faut également envisager que A puisse reconnaître le mot vide et se fier alors aux terminaux susceptibles de suivre A. Dans notre exemple cela donne :
Pour réécrire le non-terminal S on choisira la production 1 si l'entrée est a ou c ou d et la production 2 si l'entrée est b.
Pour réécrire le non-terminal A on choisira la production 3 si l'entrée est a, la production 4 si l'entrée est c et la production 5 si l'entrée est b ou d.

3.4.2 Fonctionnement de l'analyse descendante

Supposons que l'on ait une table de transition qui pour chaque non-terminal Y et mot à reconnaître en entrée m nous fournisse, si elle existe le membre droit $T(Y, m)$ d'une production applicable.

On construit un automate à pile de la manière suivante. La pile est un mot p de $(N \cup T)^*$. À l'état initial ce mot est uniquement formé du symbole de départ S. Tant que ce mot est non vide on regarde le premier caractère de cette pile, si c'est un terminal a alors on effectue la transition $a \xrightarrow{a} \epsilon$, c'est-à-dire qu'on lit un caractère sur l'entrée qui doit être égal au sommet de pile qui est dépilé. Si le sommet de pile est un non terminal X alors on cherche la production $X := \beta$ applicable en fonction de l'entrée m et on effectue la transition $X \xrightarrow{\epsilon} \beta$ où les caractères de β sont empilés en partant du dernier.

Le mécanisme décrit ne convient pas pour un analyseur syntaxique car le choix de la transition dépend a priori du mot en entrée or il y a une infinité de mots possibles. En pratique on cherchera à trouver la bonne transition simplement en regardant un ou deux caractères du mot d'entrée. Les grammaires LL(k) que nous définirons ensuite sont celles qui permettent de construire une table de transition déterministe en considérant les k premiers caractères de l'entrée.

3.4.3 Écriture fonctionnelle d'un analyseur descendant

Si on suppose donnée une fonction `expanse` qui étant donnée un non-terminal et un mot d'entrée fournit la partie droite de la production à appliquer alors il est aisé d'écrire un analyseur syntaxique descendant pour le langage sans avoir recours à l'interprétation d'un automate.

Pour cela on introduit une fonction pour chaque non-terminal X de la grammaire. Cette fonction (que l'on notera également X) prend en entrée la chaîne à reconnaître m et renvoie un mot m' tel qu'il existe un mot p tel que $m = pm'$ et $X \stackrel{*}{\Rightarrow} p$. La fonction X consomme donc une partie de l'entrée correspondant à un mot dérivable à partir de X. Ces fonctions utilisent une fonction auxiliaire `reconnaitre` plus générale qui étant donnés un mot β de $(N \cup T)^*$ et une entrée m renvoie un mot m' tel qu'il existe un mot p tel que $m = pm'$ et $\beta \stackrel{*}{\Rightarrow} p$.

Chaque fonction est programmée ainsi :

```
fonction X(m) = soit beta = expanse(X,m) dans reconnaitre(beta,m)
fonction reconnaitre(beta,m) =
         si beta = mot_vide alors m
         sinon soit a = tete(beta) et beta'=reste(beta)
               si a est terminal alors si a=tete(m) alors reconnaitre(beta',reste(m))
                                       sinon erreur
               si a est non-terminal alors soit m'=a(m) dans reconnaitre(beta',m')
```

L'analyse réussit si la fonction $S(m)$ n'échoue pas et renvoie le mot vide.

3.4.4 Gestion des erreurs

Si la grammaire est réduite alors l'erreur se produit au plus tôt c'est-à-dire que soit m_1 la partie de l'entrée déjà lue, il existe m_2 tel que m_1m_2 soit reconnu par la grammaire. En effet si p est l'état de la pile au moment de l'échec alors on a une dérivation $S \to m_1p$, chaque non-terminal X de p est productif on peut donc lui associer un mot m_X reconnu à partir de X. En substituant m_X à chaque X de p, on obtient un mot du langage.

Un gestionnaire d'erreurs pourra essayer de lire des caractères d'entrée jusqu'à trouver un caractère typique dans une analyse (par exemple le point virgule de transition entre des expressions ou un mot-clé de fin d'expression) afin de pouvoir continuer l'analyse. Ceci permet de détecter plusieurs erreurs de syntaxes lors de la même phase de compilation.

3.5 Analyse ascendante

Dans ce paragraphe on supposera les grammaires réduites et que le symbole de départ S n'apparaît que dans une seule production uniquement dans la partie gauche.

Toutes les dérivations $\alpha \stackrel{*}{\Rightarrow} \beta$ seront par défaut des dérivations droites (on réduit le non-terminal le plus à droite).

3.5.1 Fonctionnement général

L'analyse ascendante lit l'entrée de gauche à droite et reconnaît des membres droits de productions pour construire l'arbre syntaxique à partir des feuilles jusqu'à la racine qui correspond au symbole de départ.

Ces analyseurs fonctionnent de la manière suivante. On a une pile formée de mots de $(N \cup T)^*$. Les deux actions possibles sont :

- empiler un terminal de l'entrée (à la fin du mot), on dit que l'on fait une *opération de lecture* (`shift` en anglais),
- reconnaître au sommet de la pile (à la fin du mot) la partie droite d'une production et la transformer en le non-terminal correspondant, on dit que l'on fait une *opération de réduction* (`reduce` en anglais).

Dans l'état initial la pile est vide. L'automate évolue tant qu'une action peut se produire (lecture ou réduction). L'automate s'arrête lorsqu'il n'y a plus d'action possible. Si l'entrée est totalement lue et l'état de la pile est le symbole de départ S alors le mot a été reconnu, dans le cas contraire il y a échec.

Dans le cas de succès si on note α_i l'état de la pile et m_i l'état de l'entrée à l'instant i alors la suite $\alpha_i m_i$ est l'inverse d'une dérivation droite de S à $m = m_0$.

3.5.2 Les manches

Soit un mot α de $(N \cup T)^*$, un *manche* de α est une production $X{:}{=}\ \beta$ et une décomposition de α en $\alpha_1 \beta m$ avec $m \in T^*$ tels que $S \overset{*}{\Rightarrow} \alpha_1 X m \rightarrow \alpha$.
Ce manche représente une étape d'une dérivation droite de α.

Lorsqu'on cherche une dérivation droite, il n'y a besoin de reconnaître les parties droites de production qu'au sommet de la pile, en effet si on regarde deux dérivations élémentaires successives dans une réduction droite alors on a une des deux situations suivantes : La première dérivation utilise une production $X{:}{=}\ \beta Y n$ dont la partie droite contient au moins un non-terminal Y qui sert à la dérivation suivante :

$$S \overset{*}{\Rightarrow} \alpha X m \Rightarrow \alpha \beta Y n m \Rightarrow \alpha \beta \gamma n m$$

La première dérivation utilise une production $X{:}{=}\ n$ dont la partie droite ne contient que des terminaux. La dérivation suivante utilise un non-terminal présent dans la partie gauche du mot.

$$S \overset{*}{\Rightarrow} \alpha_1 Y p X m \Rightarrow \alpha_1 Y p n m \Rightarrow \alpha_1 \gamma p n m$$

On voit que entre la reconnaissance du manche $Y{:}{=}\ \gamma$ et celle du manche $X{:}{=}\ \beta Y n$ ou $X{:}{=}\ n$, il faut éventuellement lire des entrées mais qu'il n'est pas besoin d'en dépiler.

Il n'est pas toujours évident de reconnaître si le sommet de la pile est un manche.

Exemple

$$\begin{array}{ll} S & ::= aABe \\ A & ::= Abc \\ A & ::= b \\ B & ::= d \end{array}$$

Soit à reconnaître le mot *abbcde*. La première stratégie fonctionne, la seconde aboutit à un échec :

pile	entrée
ϵ	$abbcde$
a	$bbcde$
ab	$bcde$
aA	$bcde$
aAb	cde
$aAbc$	de
aA	de
aAd	e
aAB	e
$aABe$	ϵ
S	ϵ

succès

pile	entrée
ϵ	$abbcde$
a	$bbcde$
ab	$bcde$
aA	$bcde$
aAb	cde
aAA	cde
$aAAc$	de
$aAAcd$	e
$aAAcB$	e
$aAAcBe$	ϵ

echec

L'analyse ascendante est puissante et permet de traiter de nombreuses grammaires, cependant le calcul des tables de transitions est complexe et sera en général réalisé par des outils spécialisés tels que Yacc.

3.5.3 Conflits

Il peut y avoir des conflits entre lecture et réduction ainsi que des conflits entre deux réductions applicables.

Soit la grammaire des expressions arithmétiques :

S	:= E
E	:= E + E
E	:= E * E
E	:= (E)
E	:= id

Cette grammaire est ambiguë, l'expression id+id*id admet deux arbres de dérivations syntaxiques différents.

Examinons une première analyse possible :

	pile	entrée
1	ϵ	$id + id * id$
2	id	$+id * id$
3	E	$+id * id$
4	$E+$	$id * id$
5	$E + id$	$*id$
6	$E + E$	$*id$
7	$E + E*$	id
8	$E + E * id$	ϵ
9	$E + E * E$	ϵ
10	$E + E$	ϵ
11	E	ϵ
12	S	ϵ

succès

À l'étape 7, au lieu de faire une lecture, il est possible de faire une réduction on a alors la fin de l'analyse suivante :

	pile	entrée
7	$E + E$	$*id$
8	$E*$	id
9	$E * id$	ϵ
10	$E * E$	ϵ
11	E	ϵ
12	S	ϵ

succès

La première dérivation correspond au parenthésage $E+(E*E)$ tandis que la seconde correspond à $(E + E) * E$.

Les notions de précédence permettent de choisir dans certaines situations si l'analyseur doit effectuer une réduction ou une lecture. On peut associer des précédences aux non-terminaux et aux règles (par défaut, la précédence d'une règle est celle du terminal le plus à droite dans la partie droite de la production). Lors d'un conflit entre une réduction sur une production $X := \beta$ et une lecture sur a on regarde les précédences relatives de la règle et de a. Si a a une précédence strictement plus grande alors on effectue une lecture (l'opérateur a associe plus étroitement que l'opérateur de la règle β), si a a une précédence strictement plus faible alors on effectue une réduction. Dans le cas de précédences égales, ce sont les règles d'associativité qui jouent :

un opérateur associatif à gauche favorisera la réduction par rapport à la lecture, un opérateur associatif à droite choisira la lecture plutôt que la réduction.

Ces règles de précédence sont très utiles pour garder des grammaires naturelles tout en contrôlant la dérivation. Elles doivent être manipulées avec parcimonie et réservées au cas où les conflits sont bien compris.

3.6 Grammaires LL(1)

Nous revenons maintenant à l'analyse descendante et à la construction des tables de transition. Dans la suite on considère fixée une grammaire $G = (N, T, S, P)$.

3.6.1 Définitions

On peut remarquer que si $X ::= \beta$ est la transition à appliquer alors le caractère d'entrée est soit un premier caractère d'un mot dérivable à partir de β soit si β peut dériver le mot vide, un caractère qui peut suivre le non-terminal X.

Ceci nous amène à définir en général l'ensemble des premières lettres des mots de T^* dérivables à partir d'un mot de $(N \cup T)^*$.

Définition de $\textsc{null}(\alpha)$ Soit $\alpha \in (N \cup T)^*$, $\textsc{null}(\alpha)$ est vrai si et seulement si on peut dériver ϵ à partir de α (i.e. $\alpha \overset{*}{\Rightarrow} \epsilon$).

Définition de $\textsc{prem}(\alpha)$ Soit $\alpha \in (N \cup T)^*$ alors $\textsc{prem}(\alpha)$ est l'ensemble des terminaux a tels que on peut dériver un mot commençant par a à partir de α (ie $\exists m T^*. \alpha \overset{*}{\Rightarrow} am$).

Définition de $\textsc{suiv}(X)$ Soit $X \in N$ alors $\textsc{suiv}(X)$ est l'ensemble de tous les terminaux qui peuvent apparaître après X dans une dérivation gauche, c'est la réunion des ensembles $\textsc{prem}(\gamma)$ pour tout γ tel que il existe β tel que $S \overset{*}{\Rightarrow} \beta X \gamma$.

3.6.2 Calculs de null, du premier et du suivant

On cherche maintenant à calculer effectivement $\textsc{null}(\alpha)$, $\textsc{prem}(\alpha)$ et $\textsc{suiv}(X)$.

Calcul de points fixe La technique utilisée est le calcul du plus petit point fixe d'une équation monotone. On suppose que l'on se donne un ensemble A muni d'une relation d'ordre notée $\preceq$ (pas forcément totale) et d'un plus petit élément ε tel que $\forall a \in A. \varepsilon \preceq a$. On se donne maintenant une fonction F de A dans A et on cherche à calculer un point fixe de A c'est-à-dire a_0 tel que $F(a_0) = a_0$. On peut montrer que si F est monotone ($x \preceq y \Rightarrow F(x) \preceq F(y)$) et A est fini, alors il existe un plus petit point fixe.

En effet, comme ε est le plus petit élément, on a $\varepsilon \preceq F(\varepsilon)$ et du fait de la monotonicité de F on en déduit pour tout k : $F^k(\varepsilon) \preceq F^{k+1}(\varepsilon)$ comme A est fini, il existe k_0 tel que $F^{k_0}(\varepsilon) = F^{k_0+1}(\varepsilon)$. Il suffit de prendre $a_0 = F^{k_0}(\varepsilon)$ qui est un point fixe. Pour montrer que c'est le plus petit, soit b un autre point fixe. On a toujours $\varepsilon \preceq b$, dont on déduit par monotonicité que $a_0 = F^{k_0}(\varepsilon) \preceq F^{k_0}(b) = b$.

En pratique, on établira des équations de point fixe entre plusieurs variables à valeurs dans les booléens (l'ordre est alors que `faux` $\preceq$ `vrai`) ou bien dans des sous-ensembles d'une ensemble fini (l'ordre est alors l'inclusion ensembliste).

Calcul de null Le premier problème est de déterminer quels non-terminaux engendrent le mot vide.

X engendre le mot vide si et seulement si il existe une production X:= ϵ ou une production X:= $X_1 \ldots X_n$ où tous les X_i engendrent le mot vide.

Les équations étant mutuellement récursives il faut procéder par itérations successives. À la première itération on suppose qu'aucun des non-terminaux n'engendre le mot vide puis on modifie l'état de X s'il existe une production X:= ϵ ou une production X:= $X_1 \ldots X_n$ où tous les X_i sont déjà marqués comme produisant ϵ l'algorithme s'arrête quand deux lignes ont les mêmes informations.

Exemple Soit la grammaire :

1	E	$:= TE'$
2	E'	$:= +TE'$
3	E'	$:= \epsilon$
4	T	$:= FT'$
5	T'	$:= *FT'$
6	T'	$:= \epsilon$
7	F	$:= \texttt{id}$
7	F	$:= (E)$

À la première itération on reconnaît que E' et T' engendrent le mot vide. La deuxième itération donne le même résultat, c'est fini.

Calcul des premiers Soit une production :

$$X := m_1 X_1 \ldots m_n X_n m_{n+1}$$

Si $m_1 \neq \epsilon$ alors soit a le premier caractère de m_1 on a

$$\text{PREM}(m_1 X_1 \ldots m_n X_n m_{n+1}) = \{a\}$$

Si $m_1 = \epsilon$ alors $\text{PREM}(m_1 X_1 \ldots m_n X_n m_{n+1}) = \text{PREM}(X_1 \ldots m_n X_n m_{n+1})$. Si $\text{NULL}(X_1)$ alors :

$$\text{PREM}(X_1 \ldots m_n X_n m_{n+1}) = \text{PREM}(X_1) \cup \text{PREM}(m_2 \ldots m_n X_n m_{n+1})$$

sinon

$$\text{PREM}(X_1 \ldots m_n X_n m_{n+1}) = \text{PREM}(X_1)$$

Pour calculer les premiers on procède encore par itération. Au départ les ensembles de premier sont vides. Puis on met à jour en regardant tour-à-tour toutes les productions jusqu'à ce qu'un point fixe soit obtenu en tenant compte des non-terminaux engendrant le mot vide.

Dans notre exemple :

E	$\emptyset$	$\text{PREM}(T) = \emptyset$	$\text{PREM}(T) = \{\texttt{id}, (\}$
E'	$\emptyset$	$\{+\}$	
T	$\emptyset$	$\text{PREM}(F) = \emptyset$	$\text{PREM}(F) = \{\texttt{id}, (\}$
T'	$\emptyset$	$\{*\}$	
F	$\emptyset$	$\{\texttt{id}, (\}$	

Calcul des suivants Pour le calcul des suivants on introduit un caractère terminal $\sharp$. Ce caractère fait partie des suivants du symbole de départ.

$$\textsc{suiv}(X) = \bigcup_{Y:=\ \alpha X\beta} \textsc{prem}(\beta) \cup \bigcup_{Y:=\ \alpha X\beta, \mathrm{NULL}(\beta)} \textsc{suiv}(Y)$$

Là-aussi le calcul se fait par itérations successives en examinant pour chaque non-terminal les règles dans lesquelles il apparaît à droite.

E	$\{\sharp\}$	$\{\sharp,)\}$	$\{\sharp,)\}$
E'	$\emptyset$	$\textsc{suiv}(E) \cup \textsc{suiv}(E') = \{\sharp,)\}$	$\{\sharp,)\}$
T	$\emptyset$	$\textsc{prem}(E') \cup \textsc{suiv}(E) \cup \textsc{suiv}(E') = \{+, \sharp,)\}$	$\{+, \sharp,)\}$
T'	$\emptyset$	$\textsc{suiv}(T) \cup \textsc{suiv}(T') = \{+, \sharp,)\}$	$\{+, \sharp,)\}$
F	$\emptyset$	$\textsc{prem}(T') \cup \textsc{suiv}(T) \cup \textsc{suiv}(T') = \{*, +, \sharp,)\}$	$\{*, +, \sharp,)\}$

Exercice Faire le calcul des premiers et suivants pour les deux grammaires suivantes :

$$\left|\begin{array}{l} S{:=}\ E \\ E{:=}\ E+E \\ E{:=}\ E*E \\ E{:=}\ x \\ E{:=}\ y \\ E{:=}\ (E) \end{array}\right| \qquad \left|\begin{array}{ll} S{:=}\ E & T'{:=}\ \epsilon \\ E{:=}\ TE' & T'{:=}\ *T \\ E'{:=}\ \epsilon & F{:=}\ x \\ E'{:=}\ +E & F{:=}\ y \\ T{:=}\ FT' & F{:=}\ (E) \end{array}\right|$$

Remarques Les constructions par itérations successives, sont un cas particulier d'application du théorème de point fixe de Tarski qui dit qu'une fonction monotone sur un treillis complet admet un plus petit point fixe.

Si on cherche la plus petite solution $F(X)$ d'une équation de la forme $F(X) = B \cup F(X)$ alors il est équivalent de résoudre l'équation $F(X) = B$.

Construction d'une table de transitions à partir des calculs de premiers et de suivants Pour construire la table de transitions $T[A, a]$ qui pour un non-terminal A et une entrée a indique les expansions à effectuer, on procède ainsi. Pour chaque production $A{:=}\ \alpha$ et chaque terminal a de $\textsc{prem}(\alpha)$ on ajoute α à la case $T[A, a]$. Si $\textsc{null}(\alpha)$ alors on ajoute également α à toutes les cases $T[A, b]$ pour $b \in \textsc{suiv}(A)$ que b soit un terminal ou le symbole final $\sharp$.

Si dans la table il y a au plus une production dans chaque case, alors la grammaire est dite LL(1) (le mot est lu de gauche à droite et on construit une dérivation gauche en regardant au plus un caractère en avance).

Caractérisation des grammaires LL(1) La condition pour qu'une grammaire soit LL(1) est que pour toutes les productions $\alpha_1, \ldots, \alpha_n$ issues d'un non-terminal A et pour tout $i \neq j$:

- au plus un des α_i est tel que $\textsc{null}(\alpha_i)$,
- $\textsc{prem}(\alpha_i) \cap \textsc{prem}(\alpha_j) = \emptyset$
- si il existe i tel que $\textsc{null}(\alpha_i)$ alors pour tout j $\textsc{prem}(\alpha_j) \cap \textsc{suiv}(A) = \emptyset$

Une autre caractérisation plus abstraite des grammaires LL(1) est la suivante : Une grammaire est LL(1) si pour toutes dérivations gauches telles que :

$S \overset{*}{\Rightarrow} mY\alpha \Rightarrow m\beta\alpha \overset{*}{\Rightarrow} mx$ et $S \overset{*}{\Rightarrow} mY\alpha \Rightarrow m\gamma\alpha \overset{*}{\Rightarrow} my$, si les mots x et y ont la même première lettre alors $\beta = \gamma$.

3.6.3 Comment construire des grammaires LL(1)

Il faut souvent transformer les grammaires pour avoir une chance de les rendre LL(1).

La récursion gauche

Une grammaire qui est récursive gauche c'est-à-dire telle que l'on ait des dérivations :

$$X \stackrel{+}{\Rightarrow} X\alpha$$

ne sera jamais LL(1).

Récursion gauche directe Pour éliminer la récursion gauche on procède ainsi : On regarde pour un non-terminal A toutes les règles $A{::=}\ A\alpha_i$ et toutes les règles $A{::=}\ \beta_i$ avec β_i ne commençant pas par A. On transforme ces règles en $A{::=}\ \beta_i A'$ et $A'{::=}\ \alpha_i A'$ on ajoute une règle $A'{::=}\ \epsilon$. On a ainsi une grammaire qui reconnaît le même langage. En effet, toute dérivation gauche issue de A dans l'ancienne grammaire est de la forme :

$$A \Rightarrow A\alpha_{i_1} \Rightarrow A\alpha_{i_2}\alpha_{i_1} \Rightarrow \ldots \Rightarrow \beta_j\alpha_{i_k}\ldots\alpha_{i_1}$$

Dans la nouvelle grammaire on pourra l'écrire :

$$A \Rightarrow \beta_j A' \Rightarrow \beta_j\alpha_{i_k}A' \Rightarrow \ldots \Rightarrow \beta_j\alpha_{i_k}\ldots\alpha_{i_1}$$

Récursion gauche indirecte Cela ne suffit pas car on peut avoir des récursions gauches cachées :

$$\begin{array}{llcll} S & ::= Aa & \qquad & A & ::= Ac \\ S & ::= b & & A & ::= Sd \end{array}$$

Pour cela on procède ainsi. On ordonne les non-terminaux $A_1, \ldots, A_n$. On va procéder en n étapes. À l'étape i on veut assurer que pour toutes les productions $A_k{::=}\ A_l\alpha$ avec $k \leq i$ on a $l > k$. Cela fonctionnera s'il n'y a pas de dérivation vers le mot vide ni de cycle A se dérivant vers A.

Pour $i = 1$ il suffit d'éliminer la récursion gauche immédiate en introduisant un nouveau non-terminal A'_1 (qui lui peut contenir des ϵ-productions).

Pour une étape i, tant qu'il y a une production $A_i{::=}\ A_j\alpha$ avec $j < i$, on regarde toutes les productions courantes $A_j{::=}\ \beta$ et on remplace $A_i{::=}\ A_j\alpha$ par $A_i{::=}\ \beta\alpha$. Si β commence par un A_k alors $k > j$ et donc le processus s'arrête lorsque l'on n'a plus que des productions $A_i{::=}\ A_j\alpha$ avec $j \geq i$. Il suffit alors d'enlever les récursions gauches immédiates en introduisant un nouveau non-terminal A'_i.

Si on veut appliquer cela à la grammaire précédente. On commence par S on garde les règles

$$S{::=}\ Aa \qquad S{::=}\ b$$

Pour A on substitue Aa et b dans $A{::=}\ Sd$ ce qui nous donne :

$$A{::=}\ Ac \qquad A{::=}\ Aad \qquad A{::=}\ bd$$

On élimine la récursion gauche ce qui nous donne :

$$\begin{array}{lll} & A' & ::= cA' \\ A{::=}\ bdA' & A' & ::= adA' \\ & A' & ::= \epsilon \end{array}$$

Factorisation gauche

Une autre cause de conflit dans les productions est lorsque plusieurs productions ont une partie droite qui commence par le même terminal. Si on a n règles $A ::= a\alpha_1, \ldots A ::= a\alpha_n$, on peut les factoriser en introduisant un nouveau non-terminal et en remplaçant les règles par : $A ::= aA'$ et $A' ::= \alpha_1, \ldots A' ::= \alpha_n$.

Il peut arriver que deux productions aient des ensembles de premiers non disjoints sans avoir un préfixe commun. Pour faire apparaître des factorisations il peut être intéressant de transformer les grammaires en substituant certaines productions. Si (N, T, S, P) est une grammaire, soit $Y ::= \beta X \gamma$ une production avec $X \neq Y$ on ne change pas le langage reconnu en modifiant l'ensemble de productions de la manière suivante :

- supprimer la production $Y ::= \beta X \gamma$
- ajouter toutes les productions $Y ::= \beta\alpha\gamma$ pour tout α tel que $X ::= \alpha \in P$.

Langages non LL(1)

Il est souvent possible de transformer la grammaire d'un langage en une grammaire équivalente qui soit LL(1). Cependant ce n'est pas toujours possible comme le montre l'exemple suivant qui reconnaît le langage $\{a^n 0 b^n | n \in \mathbb{N}\} \cup \{a^n 1 b^{2n} | n \in \mathbb{N}\}$:

$$\begin{array}{lllllllll} S & ::= & A & A & ::= & aAb & B & ::= & aBbb \\ S & ::= & B & A & ::= & 0 & B & ::= & 1 \end{array}$$

Conclusion Les analyseurs LL(1) sont les plus simples à écrire, cependant ils nécessitent d'écrire des grammaires assez éloignées des grammaires utilisées pour spécifier les classes syntaxiques.

L'analyse descendante peut se généraliser au cas LL(k) où la décision est prise en fonction des k premiers caractères de l'entrée. En pratique construire les tables avec plusieurs caractères est trop coûteux.

3.7 L'analyse LR(1)

Pour une grande classe de grammaires il est possible de construire un analyseur syntaxique ascendant. Celui-ci est efficace, traite strictement toutes les grammaires LL(1), et a un comportement satisfaisant vis-à-vis des erreurs.

3.7.1 Principe général

On a vu que l'analyse ascendante fonctionnait en décidant d'effectuer des lectures ou des réductions. Pour prendre de telles décision, on va supposer que l'on s'aide d'un automate fini construit sur un ensemble Q d'états et sur l'alphabet $(N \cup T)$. Chaque transition sur un terminal peut-être étiquetée par une indication de lecture, une indication de réduction d'une certaine production, une indication de succès ou d'échec.

Si on suppose cet automate donné, l'analyseur fonctionnera ainsi :
La pile est un mot $s_0 x_1 s_1 \ldots x_n s_n$ avec s_i un état et $x_i \in (N \cup T)$.
Soit s l'état au sommet de la pile et a le premier caractère de l'entrée à reconnaître. On regarde l'étiquette de la transition pour l'état s et l'entrée a. Si c'est un succès ou un échec on arrête. Si c'est une lecture alors on empile l'entrée a ainsi que l'état résultat de la transition. Si c'est une réduction d'une production $A ::= \alpha$ et si α est de longueur p alors la pile est de la forme $s_0 x_1 s_1 \ldots x_n s_n \ldots x_{n+p} s_{n+p}$; on la remplace par $s_0 x_1 s_1 \ldots x_n s_n A s$ avec s état résultat de la transition à partir de l'état s_n pour le non-terminal A.

On peut remarquer que chaque état de l'automate correspond à la reconnaissance d'un préfixe d'un des mots qui forme la dérivation droite.

Tables d'actions et de déplacement En pratique, on ne travaille pas directement sur l'automate fini, mais on construit une table de transition qui se décompose en une table d'actions et une table de déplacements.

La table de déplacements a pour lignes les états et pour colonnes les non-terminaux. Chaque case comporte un état (éventuellement l'échec). On note `goto`(s, X), l'état résultat de la transition de l'état s sur le non-terminal X.

La table d'action a pour lignes les états et pour colonnes les terminaux. Chaque case `action`(s,a) pour un état s et un terminal a comporte :

- soit l'indication `shift` s' qui dit de lire un caractère et d'empiler l'état s'
- soit l'indication `reduce` $A{:=}\ \alpha$ qui dit de réduire la pile par la production $A{:=}\ \alpha$.
- soit l'indication de succès.
- soit l'indication d'échec.

On dira qu'une grammaire est LR(1) si étant donnée une dérivation droite dont on a identifié la dernière étape :

$$S \stackrel{*}{\Rightarrow} \alpha X m \Rightarrow \alpha\beta m$$

Soit une autre dérivation droite présentant le même état de pile $\alpha\beta$

$$S \stackrel{*}{\Rightarrow} \gamma Y p \Rightarrow \alpha\beta m'$$

Si m et m' ont le même premier caractère alors

$$\alpha = \gamma \qquad X = Y \qquad p = m'$$

Pour de telles grammaires, il est possible de construire un ensemble d'états et une table d'actions et de déplacement permettant l'analyse ascendante du langage.

3.7.2 Utilisation d'items simples

On va montrer que l'on peut construire un automate à partir des items simples déjà introduits. Cette construction est efficace mais ne sera pas suffisante pour analyser toutes les grammaires LR(1).

On se place dans le cas où le symbole de départ n'apparaît que dans la partie droite d'une seule règle $S{:=}\ \alpha\sharp$ avec $\sharp$ un terminal indiquant la fin de l'entrée qui n'apparaît que dans cette production.

L'idée est la suivante. On considère les items de G qui sont les triplets $[Y \rightarrow \alpha.\beta]$ tels que $Y{:=}\ \alpha\beta$ soit une production de la grammaire.

Soit une grammaire (N, T, S, P), on considère l'automate fini étiqueté par $(N \cup T)^*$, dont les transitions sont :

$$\begin{array}{lcl}
[Y \rightarrow \alpha.a\beta] & \xrightarrow{a} & [Y \rightarrow \alpha a.\beta] \\
[Y \rightarrow \alpha.X\beta] & \xrightarrow{X} & [Y \rightarrow \alpha X.\beta] \\
[Y \rightarrow \alpha.X\beta] & \xrightarrow{\epsilon} & [X \rightarrow .\gamma]\ \texttt{si}\ X{:=}\ \gamma \in P
\end{array}$$

et dont l'état initial est l'unique item $[S \rightarrow .\alpha]$.

Dans chaque état de cet automate, on a reconnu un mot qui est un préfixe d'un mot apparaissant dans une dérivation droite.

On va déterminiser cet automate. Pour cela on regroupe tous les états reliés par des ϵ-transitions. Cela amène à définir la notion de clôture d'un ensemble d'items.

La *clôture d'un ensemble d'items I* est définie comme le plus petit ensemble tel que :

– $I \subseteq \mathsf{cloture}(I)$
– Si $[Y \to \alpha.X\beta] \in \mathsf{cloture}(I)$ et $X{:}{=}\ \gamma$ est une production alors $[X \to .\gamma] \in \mathsf{cloture}(I)$

Les transitions de l'automate précédent déterminisé deviennent :

$$\begin{array}{lcl} I & \xrightarrow{a} & \mathsf{cloture}(\{[Y \to \alpha a.\beta] \mid [Y \to \alpha.a\beta] \in I\}) \\ I & \xrightarrow{X} & \mathsf{cloture}(\{[Y \to \alpha X.\beta] \mid [Y \to \alpha.X\beta] \in I\}) \end{array}$$

L'état initial étant la clôture de $[S \to .\alpha\sharp]$.

Analyseur LR(0)

On construit d'abord l'automate fini déterministe précédent ; cela nous donne un ensemble d'états $I_0, \dots I_n$. Les transitions déterminent la table des déplacements goto. On a

$$\mathsf{goto}(I, X) = J \qquad \text{si et seulement si} \qquad I \xrightarrow{X} J$$

Il faut maintenant décider de la table des actions.

Si on est dans un état I et que $X \to \alpha.a\beta \in I$ et $I \xrightarrow{a} J$ alors on peut lire une entrée a, l'automate devra se retrouver dans l'état J.

Pour réduire il faut avoir reconnu une partie droite de production. Il faut donc être dans un état I qui contient un item $[X \to \gamma.]$

On construit donc la table d'actions de la manière suivante :

– $\mathtt{action}(I, \sharp) = \mathtt{succès}$ s'il existe une transition $I \xrightarrow{\sharp} J$. On remarquera que cette transition se fait nécessairement vers un état réduit à $[S \to \alpha\sharp.]$
– $\mathtt{action}(I, a) = \mathsf{shift}(J)$ si $a \neq \sharp$ et $I \xrightarrow{a} J$
– $\mathtt{action}(I, a) = \mathsf{reduce}(X{:}{=}\ \gamma)$ si $X \to \gamma. \in I$.

La grammaire est LR(0) lorsqu'il n'y a pas de conflit entre deux actions (soit deux actions de réduction, soit une action de réduction et une de lecture). On remarquera qu'il n'y a pas de condition sur a pour l' introduction d'une action de réduction. Lorsqu' un état contient un item terminal, la réduction se fait quel que soit le caractère d'entrée.

Exemple La grammaire suivante est LR(0) :

0	S'	:= S $\sharp$
1	S	:= (L)
2	S	:= x
3	L	:= S
4	L	:= L,S

Les états numérotés de 1 à 9 sont formés des ensembles d'items suivants :

$$s_1 : \boxed{\begin{array}{l}[S'\to.S\sharp\ ]\\ [S\to.(L)]\\ [S\to.x\quad]\end{array}} \quad s_2 : \boxed{[S\to x.]} \quad s_3 : \boxed{\begin{array}{l}[S\to(.L)\ ]\\ [L\to.S\quad]\\ [L\to.L,S]\\ [S\to.(L)\ ]\\ [S\to.x\quad]\end{array}} \quad s_4 : \boxed{[S'\to S.\sharp]}$$

$$s_5 : \boxed{\begin{array}{l}[S\to(L.)\ ]\\ [L\to L.,S]\end{array}} \quad s_6 : \boxed{[S\to(L).]} \quad s_7 : \boxed{[L\to S.]} \quad s_8 : \boxed{\begin{array}{l}[L\to L,.S]\\ [S\to.(L)\ ]\\ [S\to.x\quad]\end{array}} \quad s_9 : \boxed{[L\to L,S.]}$$

La table d'actions et de déplacements obtenue est la suivante :

	(	)	x	,	$\sharp$	S	L
1	shift 3		shift 2			goto 4	
2	reduce 2	reduce 2	reduce 2	reduce 2	reduce 2		
3	shift 3		shift 2			goto 7	goto 5
4					succès		
5		shift 6		shift 8			
6	reduce 1	reduce 1	reduce 1	reduce 1	reduce 1		
7	reduce 3	reduce 3	reduce 3	reduce 3	reduce 3		
8	shift 3		shift 2			goto 9	
9	reduce 4	reduce 4	reduce 4	reduce 4	reduce 4		

Exercice Montrer que la grammaire suivante est LR(0).

$$\begin{array}{lllllllll} S & := & A & A & := & aAb & B & := & aBbb \\ S & := & B & A & := & 0 & B & := & 1 \end{array}$$

Construction de l'analyseur SLR(1)

La construction précédente engendre des conflits très aisément. Par exemple sur la grammaire :

0	S	:= E $\sharp$
1	E	:= T + E
2	E	:= T
3	T	:= x

On va chercher à limiter les réductions.

Le problème est de décider pour quelle entrée a effectuer la réduction. La dérivation cherchée aura la forme :

$$S \stackrel{*}{\Rightarrow} \alpha X m \Rightarrow \alpha \gamma m$$

L'entrée commence par le premier terminal de m qui est forcément dans l'ensemble $\textsc{suiv}(X)$. On ajoute donc :

$$\texttt{action}(I, a) = \texttt{reduce}(X{:=}\ \gamma)$$

si $[X \to \gamma.] \in I$ seulement si $a \in \textsc{suiv}(X)$.

La grammaire est SLR(1) si la table de transition ainsi obtenue n'a pas de conflit.

Exemple Dans l'exemple donné précédemment, on obtient les états et la table de transitions suivante :

$s_1 : \boxed{\begin{array}{l}[S\to .E\sharp\quad]\\ [E\to .T+E]\\ [E\to .T\quad]\\ [T\to .x\quad]\end{array}}$ $s_2 : \boxed{[S\to E.\sharp]}$ $s_3 : \boxed{\begin{array}{l}[E\to T.+E]\\ [E\to T.\quad]\end{array}}$ $s_4 : \boxed{\begin{array}{l}[E\to T+.E]\\ [E\to .T+E]\\ [E\to .T\quad]\\ [T\to .x\quad]\end{array}}$ $s_5 : \boxed{[T\to x.]}$ $s_6 : \boxed{[E\to T+E]}$.

	x	$+$	$\sharp$	E	T
1	shift 5			goto 2	goto 3
2			succès		
3		shift 4	reduce 2		
4	shift 5			goto 6	goto 3
5		reduce 3	reduce 3		
6			reduce 1		

Le fait d'inscrire une réduction dès que l'entrée est un suivant du non-terminal peut engendrer des conflits bien que la grammaire soit LR(1) comme le montre l'exemple suivant :

S	:= E
E	:= L=R
E	:= R
L	:= *R
L	:= `id`
R	:= L

Dans l'état initial on a les items :

$$[S \to .E], [S \to .L = R], [S \to .R], [R \to .L], [S \to .\mathtt{id}] \text{ et } [S \to . * R]$$

Par la lecture de L on arrive dans l'état formé des items :

$$[E \to L. = R] \text{ et } [R \to L.]$$

Comme $= \in \textsc{suiv}(R)$ on a un conflit entre une opération de lecture et une opération de réduction.

3.7.3 Le cas LR(1) général

On va mettre plus d'informations dans les items de manière à mieux contrôler les caractères qui peuvent arriver après.

On introduit des 1-items de la forme $[X \to \alpha.\beta, a]$ qui correspondent à : "chercher à dériver $\alpha\beta$ à partir de X le caractère suivant étant a".

Dans l'état initial on a l'item $[S \to .\alpha, \sharp]$.

Les transitions de cet automate sont :

$$\begin{array}{lcl} [Y \to \alpha.a\beta, b] & \xrightarrow{a} & [Y \to \alpha a.\beta, b] \\ [Y \to \alpha.X\beta, b] & \xrightarrow{X} & [Y \to \alpha X.\beta, b] \\ [Y \to \alpha.X\beta, b] & \xrightarrow{\epsilon} & [X \to .\gamma, c] \text{ si } X ::= \gamma \in P \text{ et } c \in \textsc{prem}(\beta b) \end{array}$$

On construit les tables de transitions de manière analogue, cependant on introduit une action de réduction pour $\mathtt{action}(I, a)$ seulement dans le cas où I comporte un objet $X \to \gamma., a$.

Pour ne pas alourdir les notations on factorise dans le même item les items qui ne diffèrent que par leur caractère d'avance. On écrira donc $[X \to \alpha.\beta, a_1a_2 \dots a_p]$ pour indiquer que les caractères suivants sont l'un des a_i.

Les transitions s'écrivent alors :

$$\begin{array}{lcl} [Y \to \alpha.a\beta, b_1 \dots b_p] & \xrightarrow{a} & [Y \to \alpha a.\beta, b_1 \dots b_p] \\ [Y \to \alpha.X\beta, b_1 \dots b_p] & \xrightarrow{X} & [Y \to \alpha X.\beta, b_1 \dots b_p] \\ [Y \to \alpha.X\beta, b_1 \dots b_p] & \xrightarrow{\epsilon} & [X \to .\gamma, \bigcup_{i=1\dots p} \textsc{prem}(\beta b_i)] \text{ si } X ::= \gamma \in P \end{array}$$

Exemple On construit l'automate LR(1) de la grammaire précédente.

Les états sont :

$$s_1 : \boxed{\begin{array}{l}[S\rightarrow .E, \sharp\ ]\\ [E\rightarrow .L = R, \sharp]\\ [E\rightarrow .R, \sharp\ ]\\ [L\rightarrow . * R, = \sharp]\\ [L\rightarrow .id, = \sharp\ ]\\ [R\rightarrow .L, \sharp\ ]\end{array}} \quad s_2 : \boxed{[S\rightarrow E., \sharp]} \quad s_3 : \boxed{\begin{array}{l}[E\rightarrow L. = R, \sharp]\\ [R\rightarrow L., \sharp\ ]\end{array}} \quad s_4 : \boxed{[E\rightarrow R., \sharp]}$$

$$s_5 : \boxed{\begin{array}{l}[E\rightarrow L = .R, \sharp]\\ [R\rightarrow .L, \sharp\ ]\\ [L\rightarrow . * R, \sharp\ ]\\ [L\rightarrow .id, \sharp\ ]\end{array}} \quad s_6 : \boxed{\begin{array}{l}[L\rightarrow *.R, = \sharp\ ]\\ [R\rightarrow .L, = \sharp\ ]\\ [L\rightarrow .id, = \sharp\ ]\\ [L\rightarrow . * R, = \sharp]\end{array}}$$

$$s_7 : \boxed{[L\rightarrow *R., = \sharp]} \quad s_8 : \boxed{[R\rightarrow L., = \sharp]} \quad s_9 : \boxed{[E\rightarrow L = R., \sharp]} \quad s_{10} : \boxed{[L\rightarrow id., = \sharp]}$$

La table de transitions est la suivante :

	id	$*$	$=$	$\sharp$	E	L	R
1	shift 10	shift 6			goto 2	goto 3	goto 4
2				succès			
3			shift5	reduce6			
4				reduce3			
5	shift 10	shift 6				goto 3	goto 9
6	shift 10	shift 6				goto 3	goto 7
7			reduce4	reduce4			
8			reduce6	reduce6			
9				reduce2			
10				reduce5			

Les grammaires LR(1) se généralisent au cas LR(k) où les items correspondent à (au plus) k caractères suivants de l'item.

3.7.4 Le cas LALR(1)

La méthode précédente fonctionne pour toutes les grammaires LR(1) cependant elle construit des tables qui sont gigantesques.

Les états formés de $1 - items$ dans l'analyse LR(1) correspondent à des états de l'analyse LR(0) mais dans lesquels chaque item a été annoté avec un liste de symboles suivants possibles. On peut avoir un état de $0 - items$ qui est dupliqué en plusieurs états de l'analyse LR(1).

En pratique le nombre d'états de l'automate canonique LR(1) est beaucoup plus gros que celui de l'analyse LR(0) ou SLR(1). On va donc revenir à la structure des états de l'analyse LR(0) mais en mettant ensemble tous les états qui partagent la même structure LR(0) sous-jacente. On se contentera de faire l'union des caractères suivants.

Si les états de l'automate LR(1) sont $I_1, \ldots, I_n$ on regroupe les états qui ont la même structure LR(0), on obtient des états $J_1, \ldots, J_p$. On détermine la table des déplacements comme dans le cas LR(0). Les actions de réductions sont introduites comme dans le cas LR(1).

S'il n'y a pas de conflits alors la grammaire est dite LALR(1).

Les seuls conflits engendrés sont des conflits entre deux réductions. En effet s'il y avait un conflit lecture/réduction c'est que l'on a dans le même état $X \rightarrow \gamma., a$ et $Y \rightarrow \alpha.a\beta, c$ mais un item $Y \rightarrow \alpha.a\beta, d$ se trouvait déjà dans l'état de l'automate LR(1) dont on est parti, d'où une contradiction.

Les grammaires LALR(1) sont moins puissantes que les grammaires LR(1) cependant le fait qu'elles soient construites sur des automates plus petits les rend plus efficaces. L'outil Yacc de génération d'analyseurs syntaxiques ascendants accepte des grammaires LALR(1). En pratique il existe des méthodes de génération de la table pour l'analyse LALR(1) qui ne nécessitent pas la construction de l'automate LR(1).

3.7.5 Liens entre les différentes classes de grammaires

Un même langage peut être reconnu par différentes grammaires. Nous avons dans ce chapitre identifié des classes de grammaires pour lesquelles il était possible de construire des analyseurs syntaxiques efficaces.

Ces classes de grammaire ont les propriétés suivantes :

- toute grammaire LL(1) est aussi LR(1),
- il existe des grammaires LL(1) qui ne sont pas LALR(1),
- toute grammaire LR(0) est SLR(1), l'inclusion est stricte,
- toute grammaire SLR(1) est LALR(1), l'inclusion est stricte,
- toute grammaire LALR(1) est LR(1), l'inclusion est stricte,
- aucune des grammaires précédemment citées n'est ambiguë.

3.7.6 Récupération des erreurs

Les grammaires LR(k) ont la propriété de s'arrêter dès que le mot reconnu ne peut pas être prolongé en un mot viable. Dans le cas de grammaires LR(0), SLR(1) ou LALR(1) il est possible que des réductions inutiles soient faites mais jamais de lecture.

Dans les cas d'erreur on cherche en général à retrouver dans la pile un non-terminal donné correspondant à une structure essentielle du programme et on cherche dans l'entrée un caractère possible pour un suivant de ce non-terminal, on ignore les caractères d'entrées jusqu'à lire celui-ci.

3.7.7 Compression des tables d'actions

Il est essentiel de compresser les tables d'actions. On peut le faire en associant à chaque état une liste des actions à effectuer en fonction du non-terminal d'entrée, on concentre en fin de liste le cas par défaut.

3.8 Analyse à partir de grammaires non contextuelles quelconques

Il est possible de construire un analyseur pour reconnaître si un mot appartient à un langage engendré par une grammaire non contextuelle. On se place dans le cas ou le langage ne comporte pas le mot vide (propriété qui est décidable). On commence par transformer la grammaire de manière à ce que toutes les productions soient de la forme $X ::= a\alpha$ avec a un terminal. On utilise ensuite le principe de l'analyseur construit à partir de l'automate à pile sur les items introduit au début de ce cours. On remarque que du fait de la forme des productions, toute expansion est suivie d'une lecture qui diminue la taille du mot d'entrée. On peut donc explorer de manière non=déterministe l'espace de recherche en étant sur que le procédé terminera.

Chapitre 4

Syntaxe abstraite

La syntaxe abstraite, par opposition à la syntaxe concrète que constitue le texte initial du programme à compiler, est la représentation interne de celui-ci par le compilateur sous la forme d'un arbre (dit arbre de syntaxe abstraite). Celui-ci apparaît lors de l'analyse syntaxique, où il est construit par les actions associées aux règles de la grammaire. Il peut ensuite prendre d'autres formes : arbre « décoré » à l'issue de l'analyse sémantique, puis code intermédiaire lors des diverses phases de production de code et d'optimisation.

L'importance de ces différents arbres de syntaxe abstraite est fondamentale, car les différentes phases du compilateur se définissent récursivement sur leurs structures. Dans ce chapitre, nous étudions la définition et l'utilisation des arbres de syntaxe abstraite sous l'angle particulier d'un compilateur écrit en Java (ou plus généralement d'un langage orienté objets).

4.1 Arbres de syntaxe abstraite

Une syntaxe abstraite peut être définie par une grammaire, analogue à celles précédemment utilisées pour définir la syntaxe concrète. Ainsi la syntaxe abstraite d'expressions arithmétiques formées d'identificateurs, de constantes entières, d'une négation unaire et de deux opérateurs binaires d'addition et de multiplication peut être ainsi définie :

$$\begin{array}{rcl} E & ::= & E + E \\ & | & E \times E \\ & | & -E \\ & | & \mathit{ident} \\ & | & \mathit{entier} \end{array}$$

Dit autrement, un arbre de syntaxe abstraite est soit une feuille étiquetée par un identificateur ou une constante entière, soit un nœud possédant un unique fils et représentant une négation, ou soit un nœud binaire étiqueté par une addition ou une multiplication.

Il est fondamental de saisir la différence entre syntaxe concrète et syntaxe abstraite. Ainsi les trois expressions concrètes

```
x+(5+y+1)*-z
x + ((5+y)+1) * (-z)
x + ((5+y+1) * (-z))
```

correspondent toutes au même arbre de syntaxe abstraite, à savoir, en notation préfixée :

$$+(x, \times(+(+(5, y), 1), -(z)))$$

La syntaxe concrète contient des espaces et des parenthèses qui n'apparaissent pas dans la syntaxe abstraite. Inversement, la syntaxe concrète « inclut » implicitement la convention que l'addition

```
abstract class Expr {}
class Add extends Expr {
    private Expr e1,e2;
    Add(Expr a1, Expr a2) { e1=a1; e2=a2; }
}
class Mult extends Expr {
    private Expr e1,e2;
    Mult(Expr a1, Expr a2) { e1=a1; e2=a2; }
}
class Neg extends Expr {
    private Expr e1;
    Neg(Expr a1) { e1=a1; }
}
class Ident extends Expr {
    private String id;
    Ident(String s) { id=s; }
}
class Num extends Expr {
    private int n;
    Num(int i) { n=i; }
}
```

FIG. 4.1 – Arbres de syntaxe abstraite en Java

est associative à gauche, notion qui disparaît de la syntaxe abstraite où la forme de l'arbre définit à elle seule le sens de l'expression.

En programmation orientée objets, la manière naturelle de définir des classes pour représenter des arbres de syntaxe abstraite consiste à introduire une classe abstraite pour chaque catégorie d'arbres de syntaxe abstraite (ici dans notre exemple il n'y en a qu'une, à savoir E) puis d'introduire pour chacune autant de sous-classes qu'il y a de nœuds distincts possibles. Dans notre exemple, cela donne cinq sous-classes, comme illustré en Java figure 4.1.

Pour construire l'arbre correspondant à l'expression ci-dessus, il suffit d'invoquer les constructeurs de ces différentes classes, de la manière suivante :

```
Expr e = new Add(new Ident("x"),
                 new Mult(new Add(new Add(new Num(5), new Ident("y")),
                                  new Num(1)),
                          new Neg(new Ident("z"))));
```

4.2 Programmation orientée objets et modularité

L'étape suivante consiste à définir des opérations sur la syntaxe abstraite. Dans notre exemple, on peut imaginer une opération très simple consistant à évaluer l'expression considérée sous la forme d'un entier (une table contenant les valeurs des identificateurs étant supposée donnée). La façon idiomatique de procéder consiste alors à équiper chaque classe d'une méthode `eval` sans argument et retournant un entier, de manière à évaluer une expression `e` par l'expression Java `e.eval()`. L'adaptation correspondante du code de la figure 4.1 est illustrée figure 4.2.

```
abstract class Expr {
    abstract int eval();
}
class Add extends Expr { ...
    int eval() { return e1.eval() + e2.eval(); }
}
class Mult extends Expr { ...
    int eval() { return e1.eval() * e2.eval(); }
}
class Neg extends Expr { ...
    int eval() { return -e1.eval(); }
}
class Ident extends Expr { ...
    int eval() { return Env.lookup(id); }
}
class Num extends Expr { ...
    int eval() { return n; }
}
```

FIG. 4.2 – Ajout de la méthode `eval` dans chaque classe

Cette façon naturelle de procéder dans un contexte orienté objets est adapté à l'extension de la syntaxe abstraite (ajout de nouveaux nœuds dans le futur), car le code propre à chaque nœud est regroupé dans la classe le définissant. Mais dans le cadre d'un compilateur, la syntaxe abstraite est (généralement) fixée une fois pour toutes, et cet avantage devient plutôt un inconvénient. En effet, le code de chaque opération sur la syntaxe abstraite (analyse sémantique, production de code, etc.) étant éclaté entre les différentes classes définissant celle-ci, il n'est pas possible de découper le code du compilateur selon les différentes phases (une classe pour l'analyse sémantique, une autre pour la production de code, etc.). Dit autrement, cette programmation orientée objet s'oppose à la modularité.

Bien entendu, on pourrait songer à utiliser massivement la construction `instanceof` de Java pour tester dynamiquement l'appartenance d'un arbre de syntaxe abstraite à telle ou telle sous-classe et effectuer ensuite les transtypages adéquats mais cette solution rend la programmation peu sûre (de nombreux bugs ne seront plus détectés qu'à l'exécution) et moins efficace (la construction `instanceof` implique un test dynamique).

Il existe heureusement une meilleure solution à ce problème, connue sous le nom d'idiome du *visiteur*.

4.3 Visiteurs

On souhaite donc regrouper au sein d'une même classe la définition d'une opération sur l'arbre de syntaxe abstraite. Pour cela on va utiliser la surcharge de Java et définir de multiples versions d'une même méthode, chacune prenant en argument un objet de l'une des sous-classes de `Expr`. Si on appelle `visit` cette méthode surchargée, la classe définissant l'opération réalisera donc l'interface suivante :

```
interface Visitor {
```

```
    int visit(Add e);
    int visit(Mult e);
    int visit(Neg e);
    int visit(Ident e);
    int visit(Num e);
}
```

Dès lors, définir une opération telle que l'évaluation d'une expression prendra la forme suivante :

```
class Eval implements Visitor {
    public int visit(Add e) { ... }
    public int visit(Mult e) { ... }
    public int visit(Neg e) { ... }
    public int visit(Ident e) { return Env.lookup(e.id); };
    public int visit(Num e) { return e.n; }
}
```

Malheureusement, dans le cas des nœuds qui ne sont pas des feuilles (`Add`, `Mult` et `Neg`), on ne peut pas directement appeler `visit(e.e1)` (resp. `visit(e.e2)`). En effet, la surcharge est résolue *statiquement* i.e. au moment de la compilation et, à ce moment-là, on ne peut connaître la classe de `e.e1` (resp. `e.e2`). Pour y remédier, on va transférer le contrôle à une méthode `accept` de l'objet, en lui passant le visiteur en argument, et cette méthode se chargera alors de s'appliquer le visiteur.

On commence donc par ajouter une méthode `accept` à la classe `Expr` et à chacune de ses sous-classes :

```
abstract class Expr {
    abstract int accept(Visitor v);
}
class Add extends Expr { ...
    int accept(Visitor v) { return v.visit(this); }
}
class Mult extends Expr { ...
    int accept(Visitor v) { return v.visit(this); }
}
class Neg extends Expr { ...
    int accept(Visitor v) { return v.visit(this); }
}
class Ident extends Expr { ...
    int accept(Visitor v) { return v.visit(this); }
}
class Num extends Expr { ...
    int accept(Visitor v) { return v.visit(this); }
}
```

Comme on le constate, le code de cette méthode est le même dans chacune des sous-classes. Il est cependant fondamental de le dupliquer ainsi pour que la surcharge puisse jouer. En effet, dans la classe `Add` par exemple, `this` est de la classe `Add` et donc l'appel à `v.visit(this)` correspondra bien à la méthode du visiteur `v` de la forme `int visit(Add e){...}`.

On peut alors terminer la définition de la classe `Eval` de la manière suivante :

```
class Eval implements Visitor {
    public int visit(Add e) { return e.e1.accept(this) + e.e2.accept(this); }
    public int visit(Mult e) { return e.e1.accept(this) * e.e2.accept(this); }
    public int visit(Neg e) { return -e.e1.accept(this); }
    public int visit(Ident e) { return Env.lookup(e.id); };
    public int visit(Num e) { return e.n; }
}
```

L'évaluation de `e` se fera alors par l'expression `(new Eval()).visit(e)`.

Comme on le constate, les définitions de la classe `Expr`, de ses sous-classes et du visiteur sont devenues complètement mécaniques : elles ne dépendent plus des opérations que l'on souhaite coder mais uniquement de la définition de la syntaxe abstraite telle que donnée au début de ce chapitre. Il existe d'ailleurs des outils de génération automatique de classes et de visiteurs à partir de descriptions au niveau des arbres de syntaxe abstraite, tels que classgen par exemple.

Chapitre 5

Analyse sémantique

L'analyse sémantique traite l'entrée syntaxique et la transforme en une représentation plus simple adaptée à la génération de code. Elle s'occupe également d'analyser l'entrée en particulier pour relier les utilisations des identificateurs à leur déclaration (on vérifiera que le programme respecte les règles de portée du langage) et vérifie que chaque expression a un type correct en fonction des règles du langage.

Dans ce chapitre nous étudions la gestion de la table des symboles qui sert à relier les noms manipulés dans le programme aux objets qu'ils désignent. Nous étudions ensuite le typage des programmes. Finalement nous définirons des notions de grammaires attribuées qui permettent d'attacher des valeurs aux nœuds d'un arbre de dérivation syntaxique.

5.1 Tables des symboles

5.1.1 Introduction

Les langages de programmation manipulent des *identificateurs* qui sont des symboles servant à désigner des objets (contenu d'une adresse mémoire, dans le cas des variables, morceau de code dans le cas de procédures, type ...)

La table des symboles conserve les informations sur les objets désignés par des noms dans le langage. Elle est mise à jour lorsqu'on rencontre la *déclaration* d'un nouvel identificateur. Elle est consultée lorsque qu'un identificateur est utilisé dans un programme.

La table des symboles permet de stocker pour chaque identificateur des informations associées à l'objet représenté. Celles-ci peuvent être de plusieurs natures : le type de l'objet, une position dans la liste des variables déclarées afin de calculer une adresse relative lors de la génération de code, une valeur.

Il peut y avoir plusieurs tables des symboles s'il y a plusieurs espaces de noms. On aura ainsi dans un langage objet des espaces différents pour les noms de classe et les nom de méthodes. Dans un langage comme ML, les types, les modules et les valeurs peuvent être rangés dans des tables de symboles différentes.

Un même identificateur peut servir à représenter différents objets. Ces objets peuvent être dans des tables différentes, ce qui nécessite de savoir syntaxiquement dans quelle table chercher, cet apparent conflit peut donc être très simplement levé. Un même identificateur (variable, procédure) peut également être déclaré plusieurs fois dans une même table. Lors de la compilation, il faudra connaître l'objet précis désigné par le nom. Ce sont les règles de portée du langage qui permettent de résoudre cette question.

5.1.2 Portée des identificateurs

Les programmes manipulent de nombreux identificateurs. Pour des raisons d'efficacité et afin d'améliorer la robustesse du code, les langages permettent d'indiquer syntaxiquement que certaines variables ne seront utilisables que dans une partie délimitée du code, appelée un *bloc*. Ainsi en dehors de cette partie du code, il ne sera pas nécessaire d'allouer une place en mémoire pour les objets correspondants.

Lors de l'analyse sémantique, le compilateur va s'assurer que toutes les variables utilisées ont bien été déclarées et sont bien *visibles* au moment où elles sont utilisées.

Les règles de portée des identificateurs sont spécifiques à chaque langage.

En C une variable est locale à la procédure ou globale à tout le programme. En Pascal, tout identificateur doit être déclaré avant d'être utilisé (d'où l'utilisation de l'instruction `forward` pour des procédures mutuellement récursives). Le corps d'une procédure peut utiliser des variables déclarées dans n'importe quelle procédure englobante.

En ML, tout identificateur doit avoir été préalablement déclaré dans une expression `let` *id* `=` ...sa portée est restreinte à l'expression après le `in` associé au `let` dans le cas d'une déclaration locale. Dans le cas d'une déclaration globale, l'identificateur peut être utilisé dans tout le module dans lequel il est défini (un fichier correspondant toujours à un module). En dehors du module, un identificateur exporté sera accessible mais sous un nom "qualifié" (`nom_du_module.ident`). Des directives d'ouverture de modules permettent localement d'utiliser l'identificateur sans le préfixe. Si une nouvelle déclaration venait à cacher la définition importée, celle-ci serait toujours accessible par l'intermédiaire de son nom long. Une déclaration de fonction en ML est par défaut non récursive. Il faut utiliser le mot clé `let rec` pour les fonctions récursives.

En Java, le corps d'une méthode peut utiliser d'autres méthodes de la même classe qui peuvent être définies plus loin, ceci nécessite de traiter en bloc les définitions de classes. On peut définir dans une même classe plusieurs méthodes de même nom, le type de leurs arguments permet de choisir statiquement la méthode qui s'applique. De nombreuses classes peuvent redéfinir les mêmes méthodes. Si les classes sont disjointes, cette ambiguïté de nom sera résolue par le typage. Si le même identificateur est défini dans une classe et redéfini dans l'une de ses sous-classes alors on imposera une cohérence entre les types des deux méthodes. Le choix du code à appliquer s'effectuera en général de manière dynamique à l'exécution. Il est parfois utile d'utiliser des constructions syntaxiques telles que `super` pour accéder à une méthode de la classe mère dont le nom a été recouvert lors d'une redéfinition.

5.1.3 Représentation de la table des symboles

Après l'analyse du programme, on doit pouvoir retrouver pour n'importe quel identificateur les informations associées. Ceci peut se faire en décorant l'arbre de syntaxe abstraite. Il est possible d'associer à chaque utilisation d'un identificateur un pointeur vers la partie de l'arbre correspondant à sa déclaration (qui pourra être enrichie avec des informations telle que le type). On peut aussi associer à l'utilisation de l'identificateur toute l'information utile (mais comme un identificateur est utilisé de nombreuses fois ce n'est pas forcément efficace). Une autre solution est de stocker l'information sur les objets dans une table et d'associer une adresse dans cette table à chaque utilisation de l'identificateur. Cette adresse peut être un pointeur, un entier ou bien un nom unique.

À côté de cette structure persistante, il est nécessaire de gérer une table pour la vérification de la portée des identificateurs. Cette table doit permettre à chaque instant de connaître exactement les identificateurs visibles et l'information associée. Il faut pouvoir ajouter de nouveaux identificateurs, retrouver rapidement si un identificateur est visible et pouvoir retirer un identificateur lorsque celui-ci correspond à un objet qui n'est plus accessible lors de la fermeture d'un bloc.

Supposons que l'on analyse un programme de la forme :

```
let x = (let y = 2 in y*y + 2*y +1) in x+y
```

Lorsqu'on commence à analyser cette expression on a une certaine table des symboles visibles T, pour l'analyser on repère une expression `let x = e in e'` qui déclare un nouvel identificateur `x`. Pour trouver l'information relative à cet identificateur, on analyse le corps `e` de la définition (ici `let y = 2 in y*y + 2*y +1`). On est donc amené à analyser le corps de la définition de `y` c'est-à-dire `2`. On transforme la table T en une table T' dans laquelle on a ajouté l'identificateur `y` associé par exemple à l'information de type `entier`. Cette déclaration de `y` rend invisible toute autre information sur `y` qui aurait pu se trouver dans T. En utilisant la table T' on analyse l'expression `y*y + 2*y +1` qui est de type entier. Maintenant on sait que `x` est un entier et on peut ajouter à T l'entrée `x` avec le type `entier` et procéder à l'analyse du corps de la déclaration `x+y`. L'identificateur `y` dans cette expression réfère à une déclaration de `y` qui devait déjà se trouver dans la table T. À la fin de cette analyse, les différentes utilisations des identificateurs dans l'arbre de syntaxe abstraite doivent être reliées à la bonne information de typage, la table des symboles visibles est à nouveau T.

Les tables de symboles doivent être optimisées car le nombre d'identificateurs est important et l'accès à l'information doit être rapide. On peut choisir de les implanter de manière fonctionnelle ou impérative. Dans le cas d'une représentation fonctionnelle l'analyse de visibilité pourra s'écrire :

```
fonction visible : table * asa -> asa_typé
visible (T,let(x,e,e ')) =
        soit f = visible(T,e),
             typ = type_de(T,f),
             T' = ajoute ((x,typ),T),
             f' = visible(T',e'),
             let(x:typ,f,f')
```

Ceci fonctionne si la représentation de la table T est fonctionnelle c'est-à-dire si la construction de T' par ajout de `(x,typ)` à T ne modifie pas T. Ceci n'est pas le cas si la table est représentée comme une table de hachage (sauf à recopier cette table ce qui serait très inefficace). Si on choisit une représentation par une table de hachage, il est nécessaire de retirer explicitement les identificateurs ce qui donnera, la table pouvant être globale :

```
fonction visible_imp : asa -> asa_typé
visible_imp (let(x,e,e ')) =
        soit f = visible_imp(e),
             typ = type_de(f),
             ajoute x;
             soit f' = visible_imp(e'),
             retire (x); let(x:typ,f,f')
```

Les tables de hachages avec liaison en liste des entrées correspondant à des identificateurs de même valeur de hachage se prêtent bien à l'implantation des tables de symboles visibles. En effet quand deux identificateurs portent le même nom, celui déclaré en dernier cachera naturellement les déclarations antérieures. L'ajout et la suppression s'effectuant en pile, pour retirer un identificateur, il suffit de retirer la première entrée correspondant à la valeur de hachage.

Il est possible d'introduire dans le même bloc un nombre arbitraire d'identificateurs. On doit donc à la sortie du bloc retrouver l'ensemble des identificateurs introduits dans le bloc pour les retirer de la table des symboles visibles. Ceci nécessite de conserver une pile des blocs ouverts avec pour chaque bloc la liste des identificateurs introduits. Ceci peut se faire de manière fonctionnelle en conservant une pile de liste d'identificateurs ou de manière plus impérative en liant ensemble les identificateurs d'un même bloc dans la table des symboles visibles et en gardant dans une pile l'adresse de la dernière variable introduite dans le bloc.

5.1.4 Représentation des symboles

Lorsque l'on analyse les symboles du programme, il est nécessaire de faire de nombreuses comparaisons (égalité ou inégalité si on utilise des arbres binaires de recherche). Pour cela il peut être intéressant d'utiliser une représentation plus efficace des identificateurs sous la forme d'un entier. On utilise alors une table de hachage qui à chaque chaîne associe un entier unique. Il est nécessaire de pouvoir établir la correspondance inverse. Il faut toujours être attentif au fait que cette optimisation n'apporte pas de surcoût lié à sa gestion.

5.1.5 Schéma de vérification de portée

(cf TD Portée des identificateurs)

Une manière de procéder à la vérification des portées dans un programme est la suivante :

L'entrée de l'analyse est un arbre de syntaxe abstraite dans lequel l'utilisation des identificateurs est repérée par un nom, cet arbre peut être le résultat de l'analyse syntaxique. Après l'analyse de portée, une table des déclarations du programme est créée et chaque utilisation d'un identificateur est remplacée par un index dans cette table.

La table des symboles visibles, utilisée lors de l'analyse, contient une liste d'associations entre un nom correspondant à un symbole visible et un entier représentant l'index de la déclaration correspondante dans la table des déclarations. Dans le cas d'une représentation impérative de la table des symboles visibles, il est nécessaire de conserver en plus une pile gardant la structure des identificateurs déclarés dans chaque bloc ouvert.

5.2 Types

5.2.1 Types et relations de typage

Définitions

Les types sont des expressions d'un langage qui servent à représenter non des objets ou des instructions de contrôle mais des domaines qui vont permettre de classifier les autres objets du langage.

Par exemple `int,bool`,...

Les objets du langage et les expressions de types sont reliées par une relation dite de typage qui exprime la propriété : "une expression e est correctement formée et appartient au domaine représenté par le type τ". On dira que e a le type τ et on écrira en général cette relation $e : \tau$. On dira que e est *bien typé* s'il existe un type τ tel que $e : \tau$.

Une expression du langage comporte des identificateurs auxquels il faut associer également une information de type, qui contraint l'ensemble des objets que cet identificateur peut représenter.

La relation de typage s'écrira donc en général :

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$$

qui se lit " sous l'hypothèse où x_1 a pour type τ_1, ..., x_n a pour type τ_n alors l'expression e a pour type τ".

Exemple : $x : \texttt{int} \vdash x + 3 : \texttt{int}$

Règles de typage Cette relation est souvent décrite par un système de règles d'inférence.

$$\frac{}{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i} \qquad \frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_1 : \texttt{int} \quad x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_2 : \texttt{int}}{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_1 + e_2 : \texttt{int}}$$

Profil des procédures et fonctions Les procédures et fonctions manipulées dans un programme ont un *profil* qui décrit le type des arguments attendus et dans le cas des fonctions, le type du résultat.

Dans les langages fonctionnels, les fonctions sont des objets comme les autres et donc ont un type qui décrit en particulier le type de leurs arguments et de leur résultat.

Dans les autres langages, on parle plutôt de profil de la fonction ou de la procédure. On notera $\sigma_1 \times \cdots \times \sigma_p \to \tau$ le profil d'une fonction dont le i-ème argument est de type σ_i et le résultat de type τ. Dans le cas des procédures, il est parfois commode de ne pas distinguer les notions d'expressions et de commandes et de voir une commande comme une expression dont le type est distingué `unit`.

Si f est une fonction de profil $\sigma_1 \times \cdots \times \sigma_p \to \tau$ alors la règle de typage correspondante est :

$$\frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_1 : \sigma_1 \quad x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_p : \sigma_p}{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash f(e_1, \ldots, e_p) : \tau}$$

À quoi servent les types ?

Les types ont plusieurs utilités :
- Détecter de manière précoce (statique, à la compilation) des erreurs de programmation (mauvaise instanciation de procédures, application erronée d'opérateurs).
- Résoudre des ambiguïtés dans les notations en permettant d'utiliser les mêmes conventions pour des opérations sémantiquement différentes.
  Exemple : $4+6$, $4.0+6.1$, $4+6.1$. Les opérations de la machine appropriées pourront être appelées et éventuellement des conversions seront introduites.
- Déterminer la taille de l'espace mémoire à allouer à chaque expression.

Suivant le but recherché on utilisera un système de type plus ou moins compliqué.

Différentes formes de typage

Le typage peut être :

Statique ou dynamique
- Statique : la vérification est faite à la compilation, les types peuvent ensuite être ignorés à l'exécution.
- Dynamique : les types sont calculés à l'exécution en particulier dans les langages objets pour déterminer de manière tardive la méthode à utiliser. La taille d'un tableau à allouer peut également n'être connue qu'au moment de l'exécution.

Explicite ou implicite
- Explicite : les variables sont déclarées avec leur type comme dans Pascal.
- Implicite : le compilateur est capable de déterminer l'existence d'un type sans annotation de l'utilisateur comme dans ML.

Monomorphe ou polymorphe

- Monomorphe : chaque expression a au plus un type
- Polymorphe : une expression peut avoir plusieurs types on distingue :
  - Le polymorphisme ad-hoc : la même expression a plusieurs types qui ne correspondent pas forcément à la même séquence d'exécution.
    Par exemple l'addition sur les entiers ou les flottants, les méthodes dans les langages de programmation objet.
  - Le polymorphisme paramétrique : une expression admet plusieurs types correspondant à la même séquence d'exécution.
    Par exemple : un programme de tri tel que quicksort s'écrit de la même manière qu'il trie des entiers ou des chaînes de caractères.
  - L'héritage des langages objets offre une forme de polymorphisme, lorsque la méthode n'est pas redéfinie dans les sous-classes, le même code s'applique aux objets de différentes classes, si la méthode est redéfinie, alors c'est un code différent qui va s'appliquer.

  Lorsqu'il y a plusieurs types possibles, on cherche s'il existe un *type principal* pour une relation d'ordre établie entre les types. Le type τ sera dit type principal d'une expression e dans un environnement Γ si

$$\Gamma \vdash e : \sigma \Leftrightarrow \tau \leq \sigma$$

Propriétés attendues

Parmi les propriétés attendues d'un système de typage il y a :

- Décidabilité : Soit $x_1 : \tau_1, \ldots x_n : \tau_n$ un environnement, e une expression et τ un type il est décidable de savoir si e est correctement typé de type τ dans l'environnement donné.

  En particulier dans un langage permettant de représenter toutes les fonctions récursives, on ne peut tester par le typage l'appartenance d'un indice au domaine d'un tableau ou la terminaison d'une fonction.
- Inférence : Soit e une expression, il est décidable de savoir s'il existe un environnement $x_1 : \tau_1, \ldots x_n : \tau_n$ et un type τ tel que e est correctement typé de type τ dans l'environnement $x_1 : \tau_1, \ldots x_n : \tau_n$.
- Efficacité : l'algorithme de vérification de type ne doit pas prendre trop de ressources (temps espace).
- Validité : un type est une propriété associée à une expression de programme, on souhaite qu'elle soit stable par rapport à l'exécution du programme et donc que l'absence d'erreur de typage à la compilation garantisse l'absence d'erreurs de typage à l'exécution.
  On demande que si e est typé de type `int` alors le résultat de l'évaluation de e (si celle-ci termine) soit une valeur entière.

5.2.2 Types et constructeurs de type

Les types des langages de programmation sont en général organisés en types de base comme les entiers flottants ou chaînes de caractères et des types structurés obtenus en combinant des types à l'aide de constructeurs de type.

- les types produits $\tau_1 * \tau_2$,
- les types record $\{lab_1 : \tau_1; \ldots; lab_n : \tau_n\}$ qui permettent de représenter des produits avec un accès nommé aux composantes,
- les types tableaux $array[n, \tau]$ qui contiennent une indication de la taille du tableau (sous forme d'une constante ou d'un paramètre) et permettent de regrouper n objets de même type τ.

Le tableau est dit statique si on connaît sa taille à la compilation et dynamique, si la taille dépend de paramètres qui ne seront connus qu'à l'exécution du programme.
- les pointeurs $\uparrow \tau$ qui désignent une adresse vers un objet de type τ.

5.2.3 Égalité sur les types

L'algorithme de typage effectue de nombreux tests d'égalité entre types, certains compilateurs utilisent une représentation approchée (on ignorera par exemple le domaine des tableaux) permettant de détecter très efficacement que deux types sont différents.

Nommage de types Les langages permettent en général de donner un nom à une expression de type. La gestion du typage doit alors décider si l'égalité entre deux types nommés est l'égalité des types sous-jacent ou bien l'égalité des noms.

Le fait de se limiter à l'égalité des noms rend la détection de l'égalité plus rapide et offre une certaine forme de spécification. Si $f : name_1 \to \tau$ et $x : name_2$ alors f ne peut pas être appliquée à x même si $name_1$ et $name_2$ sont des abréviations pour des expressions de même structure. Par exemple si on choisit de représenter des jours et des mois par des entiers, cette distinction peut prévenir l'utilisation d'opérations telles que l'addition d'un jour et d'un mois. Les noms doivent alors plutôt être considérés comme des abstractions que comme des abréviations.

Dans un langage tel que ML, chaque déclaration de type construit (record, type avec constructeur) est vue comme l'introduction d'un nouvel objet même s'il a le même nom et la même structure qu'un type défini précédemment, ce qui signifie que la représentation interne devra garder une information supplémentaire pour distinguer les différentes occurrences d'une déclaration.

Certains types peuvent être définis de manière récursive, un nom définissant une structure mentionnant ce nom. Une égalité par nom évitera un bouclage dans la vérification de type. Les types peuvent alors être représentés par des graphes.

Il arrive donc qu'à une même expression puisse être associées plusieurs expressions de types qui représentent le même ensemble de valeurs. On a en général dans de tels systèmes une règle d'inférence :

$$\frac{\Gamma \vdash e : \tau \quad \tau \equiv \sigma}{\Gamma \vdash e : \sigma}$$

Pour vérifier qu'une expression est correctement formée, il convient de préciser les endroits où l'usage de cette règle est nécessaire. Par exemple au moment de l'application d'une fonction de profil $\sigma_1 \times \sigma_2 \cdots \times \sigma_p \to \sigma$ à ses arguments e_i dont le type calculé est τ_i, il faudra vérifié que chaque expression de types σ_i est équivalente à τ_i.

5.2.4 Surcharge d'opérateurs

Une expression est surchargée, si à un moment donné cette expression peut correspondre à plusieurs objets. On parle d'opérateurs surchargés (pour les opérateurs arithmétiques par exemple), mais une constante peut être également surchargée, par exemple le symbole 4 peut représenter un entier ou bien un flottant. Cette pratique est très courante en mathématique, elle est rejetée par certains langages de programmation (comme ML) car compliquant la sémantique du langage et sujette à confusion. Elle est autorisée dans des langages tels que ADA où l'ambiguïté doit être résolue de manière statique. Dans les langages objets, il y a deux formes de surcharge. La première permet de définir dans la même classe plusieurs méthodes de profil différent, l'ambiguïté est résolue de manière statique en tenant compte des classes et du nombre des arguments. La seconde consiste à définir une méthode de même nom et profil dans deux classes dont l'une est sous-classe de l'autre. Le choix de la méthode à appliquer ne se fait qu'à l'exécution en fonction de la classe de l'objet auquel la méthode s'applique. L'appel à une telle méthode surchargée est alors compilé vers un code effectuant un branchement entre différentes implantations possibles de la méthode.

Surcharge à la manière de Ada On va s'intéresser à la résolution statique de la surcharge à la manière de Ada. Chaque identificateur ou constante correspond à un ensemble fini d'objets visibles, chacun ayant un type. Étant donné une expression, celle-ci sera bien typée si on sait associer à chaque constante ou identificateur un objet unique telle que l'expression ainsi interprétée soit correctement typée.

Pour cela on utilise l'information de type, ce qui interdit a priori que le même identificateur soit associé à deux objets distincts de même type. Le type détermine donc pour un identificateur exactement l'objet concerné.

On regarde par exemple le cas d'expressions de la forme $f(e_1, \ldots, e_n)$ avec f un opérateur auquel est associé un ensemble de profils ou bien une constante c ou une variable x à qui est associée un ensemble de types.

Un profil d'opérateur est de la forme $\tau_1 \times \cdots \times \tau_n \to \tau$ ou τ_i est le type attendu du i-ème argument (on dira le i-ème *domaine* du profil et τ le type du résultat on dira l'*image* du profil, n est l'arité du profil.

L'algorithme de résolution procède en plusieurs phases de parcours de l'expression. L'expression peut-être initialement annotée avec pour chaque symbole, l'ensemble des profils associés. Les parcours vont permettre d'éliminer des types impossibles. La surcharge est résolue syntaxiquement s'il y a exactement une seule affectation possible d'un profil à un symbole, pour laquelle l'expression est correctement typée.

La première phase consiste à associer à chaque sous-expression l'ensemble de ces types possibles. Pour les feuilles qui sont des variables ou des constantes, cela nous est donné par les règles du langage. Lorsque l'expression correspond à $f(e_1, \ldots, e_n)$, l'application de l'opérateur f aux expressions e_1, ..., e_n, on calcule récursivement les types possibles pour les e_i puis on regarde parmi les profils possibles de f tous ceux de la forme $\tau_1 \times \cdots \times \tau_n \to \tau$ pour lesquels τ_i est un type possible de e_i, on aura alors que τ est un type possible de $f(e_1, \ldots, e_n)$ et le profil $\tau_1 \times \cdots \times \tau_n \to \tau$ est un profil possible de f.

Si à ce point le sommet de l'analyse admet plusieurs types possibles, la surcharge ne peut être résolue statiquement. Cependant le fait que le sommet admette un type unique n'est pas une condition suffisante pour que la surcharge soit résolue. Il faut également que chaque sous-expression ait un type unique. On procède donc à une descente dans l'arbre dans laquelle on va chercher à assigner un type unique à chaque sous-expression et un profil unique à chaque opérateur ou constante. On sait déjà que $f(e_1, \ldots, e_n)$ a un type unique τ, il faut que f n'ait qu'un seul profil possible d'image τ, à savoir $\tau_1 \times \cdots \times \tau_n \to \tau$ alors on continue de descendre dans l'arbre en utilisant τ_i comme type unique de e_i. Si à un moment donné on se trouve avec une expression $g(a_1, \ldots, a_k)$ dont le type unique est σ mais qui a deux profils possibles d'image σ c'est que le typage n'est pas suffisant pour lever l'ambiguïté liée à la surcharge.

Exemple $+ : \mathit{int} \times \mathit{int} \to \mathit{int}, \mathit{float} \times \mathit{float} \to \mathit{float}$. On suppose que les constantes telles que $3, 4$ ont les types *int* et *float*. Lorsqu'on écrit $(1+4)+3.2$ on détermine les types de $1, 4$ qui sont *int* et *float* ce qui fait que le type de $(1+4)$ est aussi *int* ou *float* puis 3.2 étant de type *float* le résultat complet est de type *float* donc $(1+4)$ est de type *float* et chaque symbole entier est de type *float* ce qui signifie probablement qu'il devra lui être appliqué une fonction de conversion.

Surcharge à la manière de Java Dans Java, pour résoudre statiquement les problèmes de surcharge, on ne regarde que les types des arguments et pas le contexte dans lequel ils sont utilisés. Par contre il faut prendre en compte le sous-typage entre classes. Ainsi lorsqu'une méthode m est appliquée à un objet de classe A avec des paramètres $e_1, \ldots, e_n$ de types $A_1, \ldots, A_n$, on cherche parmi toutes les définitions possibles de méthodes m à n-arguments dans les sur-classes de A s'il en existe exactement une plus petite qui s'applique au profil des arguments et de l'objet.

5.2.5 Inférence de type à l'aide de schémas de type

Déclarer explicitement le type de chaque variable est laborieux, alors que celui-ci peut souvent se déduire naturellement du contexte de l'utilisation de variable. Par exemple l'instruction d'initialisation $x := 1$ impose que le type de x soit entier.

On s'intéresse au problème d'inférence de type qui étant donné une expression e comportant des identificateurs $x_1, \ldots, x_n$ cherche à savoir s'il existe des types $\tau_1, \ldots, \tau_n$ et τ tels que $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$.

Un tel problème n'est pas toujours décidable. Il l'est dans le cadre d'un calcul fonctionnel simplement typé et nous allons montrer les grands schémas pour le mettre en œuvre.

Langage étudié Le langage utilisé comporte les éléments suivants :
- des constantes (notées c) dont le type est donné,
- des opérateurs primitifs (opérations arithmétiques par exemple) dont le type est donné,
- des identificateurs associés à des expressions, $x = e$
- des fonctions définies par l'utilisateur pour lesquelles seuls le nom de la fonction, les paramètres et le corps sont donnés, $(f(x_1, \ldots, x_n) = e)$ les fonctions récursives seront notées de manière spécifique ($\texttt{rec}\ f(x_1, \ldots, x_n) = e$)
- une expression est soit une constante, soit un paramètre, soit un identificateur, soit l'application d'un opérateur ou d'une fonction de l'utilisateur à des expressions,
- un programme bien formé est composé d'une suite de déclarations d'identificateurs ou de fonctions.

Les types de ce langage sont un ensemble de types de base, et les types des fonctions sont de la forme $\tau_1 \times \ldots \times \tau_n \to \tau$ avec τ_i et τ des types.

Programme bien formé On vérifie qu'un programme est bien formé et on lui associe un *environnement* qui est une suite d'association de types à des identificateurs $x_1 : \tau_1, \ldots, x_n : \tau_n$.
- Le programme composé d'aucunes déclarations correspond à un environnement vide.
- Si un programme p est bien formé et correspond à un environnement Γ :
  - si $\Gamma \vdash e : \tau$ alors le programme composé de p et de $x = e$ est bien formé et correspond à l'environnement $\Gamma, x : \tau$;
  - si $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ alors le programme composé de $\Gamma, f(x_1, \ldots, x_n) = e$ est bien formé et correspond à l'environnement $\Gamma, f : \tau_1 \times \ldots \times \tau_n \to \tau$;
  - si $\Gamma, f : \tau_1 \times \ldots \times \tau_n \to \tau, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$ alors le programme composé de $\Gamma, \texttt{rec} f(x_1, \ldots, x_n) = e$ est bien formé et correspond à l'environnement $\Gamma, f : \tau_1 \times \ldots \times \tau_n \to \tau$.

Les règles pour la formation des expressions sont celles que nous avons vues précédemment.

Inférence de type On se pose la question de vérifier qu'un programme est bien formé et de calculer l'environnement associé.

On introduit une notion de schéma de type, qui est construit comme les types du langage mais avec une notion supplémentaire de type : les *variables de type* qui seront notées α, β.

On peut substituer à une variable α dans un schéma de type σ, un schéma de type σ', on obtient ainsi un nouveau schéma de type qui est appelé *une instance* de σ. Un type est un cas particulier de schéma de type dans lequel il n'apparaît pas de variables. Un schéma de type avec variables représente un ensemble de types obtenus en substituant des types à chacune des variables.

Exemple $array[\uparrow \tau]$ est une instance de $array[\uparrow \alpha]$ qui est une instance de $array[\alpha]$.

On se pose maintenant un problème plus général que celui de l'inférence de type qui est : étant donnés des schémas de type $\sigma_1, \ldots, \sigma_n$ et σ, existe-t-il une substitution ρ de types pour les variables de type, tels que si $\tau_i = \rho(\sigma_i)$ et $\tau = \rho(\sigma)$ alors $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$.

On dit alors que ρ est une *solution au problème d'inférence contraint* par $\sigma_1, \ldots, \sigma_n$ et σ.

Soit $\sigma_1, \ldots, \sigma_n$ et σ des schémas de type, on notera $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \sigma$ si toute substitution ρ est une solution au problème d'inférence donné par $\sigma_1, \ldots, \sigma_n$ et σ.

Le problème d'inférence initialement posé est une instance du problème d'inférence contraint.

Rappel : unification On rappelle que deux expressions contenant des variable a et b sont unifiables s'il existe une substitution ρ des variables par des termes tels que $\rho(a) = \rho(b)$ et que lorsqu'il existe un unificateur, il existe un unificateur principal c'est-à-dire ρ_0 tel que $\rho_0(a) = \rho_0(b)$ et si $\rho(a) = \rho(b)$ alors il existe ρ' tel que $\rho = \rho' \circ \rho_0$.

Exercice Décrire l'ensemble des unificateurs des couples de termes suivants :
- `int` et α
- `int` et `bool`
- α et β
- α et `array`$[\beta]$

Algorithme d'inférence contraint Pour résoudre le problème d'inférence contraint pour $\sigma_1, \ldots, \sigma_n$ et σ, on calcule une substitution de schémas de types aux variables de types, telle que $x_1 : \rho(\sigma_1), \ldots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$.

On cherche de plus une solution principale, c'est-à-dire que toute autre solution est une instance de celle-ci.

On se donne donc un environnement qui associe un schéma de type σ_i à chaque variable x_i, une expression e et un schéma de type σ. On construit la substitution ρ par récurrence sur la structure de e.
- Si e est la variable x_i alors la substitution cherchée est l'unificateur principal de σ_i et σ car les substitutions ρ cherchées sont exactement celles qui vérifient $\rho(\sigma_i) = \rho(\sigma)$. L'inférence échoue si σ_i et σ ne sont pas unifiables.
- Si e est une constante de type τ alors on cherche une substitution ρ telle que $\rho(\sigma) = \tau$ qui est un cas particulier d'unification dans lequel un des membres ne contient pas de variables, appelé problème de filtrage.
- Dans le cas d'un appel de fonction $f(e_1, \ldots, e_n)$ on regarde le profil de la fonction qui est $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$, on appelle récursivement l'inférence sur l'expression e_1 avec le type attendu τ_1, si l'algorithme réussit on obtient une substitution ρ_1, on appelle alors l'algorithme sur l'expression e_2 avec l'environnement $x_1 : \rho_1(\sigma_1), \ldots, x_n : \rho_1(\sigma_n)$ on obtient alors une substitution ρ_2 on continue jusqu'à e_n dont le type attendu est τ_n dans l'environnement $x_1 : \rho_{n-1}(\ldots(\rho_1(\sigma_1))\ldots)\ldots, x_n : \rho_{n-1}(\ldots(\rho_1(\sigma_n))\ldots)$. On obtient alors une substitution ρ_n. On compare alors $\rho_n(\ldots(\rho_1(\sigma))\ldots)$ avec le type attendu τ de la fonction, s'il existe une substitution ρ telle que $\rho(\rho_n(\ldots(\rho_1(\sigma))\ldots)) = \tau$ alors la substitution résultat est $\rho o \rho_n o \ldots o \rho_1$.
- Il faut également être capable d'inférer les profils des fonctions. Si la fonction est définie par $f(x_1, \ldots, x_n) = e$, on introduit de nouvelles variables de type $\alpha_1, \ldots, \alpha_n$ pour chaque variable x_i et α pour le résultat, puis on applique l'algorithme d'inférence de type à e. La substitution résultat nous fournit le type des x_i et du résultat.
- Si on a une fonction récursive on introduit de plus $f : \alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha$.

La solution à l'inférence de type est unique si la substitution résultat ρ est telle que $\rho(\sigma_i)$ et $\rho(\sigma)$ sont des types (pas de variables de type).

Exemple

```
f(x)=x
g(x,y)= y or f(x)>3
```

5.2.6 Polymorphisme

Dans le cas précédent on s'est restreint à la situation où l'analyse nous donnait une solution unique pour le typage, ce qui est restrictif. En fait on peut au lieu d'associer un type unique à chaque expression, associer un schéma de type unique qui représente une famille de types. C'est ce qui est fait dans les langages de la famille ML.

Par exemple, la fonction $f(x) = x$ a pour type $\tau \to \tau$ pour tout type τ. On lui associe le schéma de type $\alpha \to \alpha$.

L'algorithme d'inférence vu précédemment peut s'adapter aisément à l'inférence de schémas de type.

5.2.7 Le cadre du λ-calcul simplement typé

Le problème de l'inférence de type peut être posé dans un cadre très général mais conceptuellement très simple qui est celui du λ-calcul simplement typé. Dans ce langage, il n'y a pas de distinction entre les valeurs de base telles que les entiers et les valeurs fonctionnelles.
Les types sont :

- soit des types de base (entiers, booléens, unit ...) ;
- soit des constructeurs de types appliqués à des types ($\tau_1 * \tau_2$, τ `list` ...) ;
- soit un type fonctionnel $\tau_1 \rightarrow \tau_2$, le type des fonctions qui prennent un argument de type τ_1 et calculent un résultat de type τ_2.

Les expressions sont :

- soit des variables (introduites comme paramètre de fonction ou bien déclarées par l'utilisateur) ;
- soit des constantes primitives (qui peuvent être fonctionnelles) ;
- soit l'application $(f\ e)$ d'une expression f (représentant une fonction) à une expression e (l'argument) ;
- soit une expression abstraite par rapport à une variable $\texttt{fun}\ x \rightarrow e$ qui représente la fonction qui à toute valeur v pour le paramètre x associe le résultat de l'évaluation de e dans lequel x a été remplacé par v.

Un programme est une suite de déclarations, chaque déclaration est de la forme :
$\texttt{let}\ \mathrm{x} = e$
Ce formalisme minimal est suffisant pour décrire les principales caractéristiques des langages de programmation. Ainsi la déclaration "classique" de fonction :
$\texttt{let}\ f(x_1 : \tau_1; \ldots ; x_n : \tau_n) : \tau = e$
se traduit dans ce formalisme par la déclaration : $\texttt{let}\ f\ = \texttt{fun}\ x_1 \rightarrow \cdots \rightarrow \texttt{fun}\ x_n \rightarrow e$. qui a pour type $\tau_1 \rightarrow\ \cdots \rightarrow \tau_n \rightarrow \tau$.
Cette traduction n'est valide que si f n'est pas récursive. Une fonction récursive se traduira en utilisant un opérateur de point fixe `fix`. On écrira alors : $\texttt{let}\ f\ = \texttt{fix}\ (\texttt{fun}\ f \rightarrow \texttt{fun}\ x_1 \rightarrow \cdots \rightarrow \texttt{fun}\ x_n \rightarrow e)$
Si F a pour type $\phi \rightarrow \phi$ alors $(\texttt{fix}\ F)$ a pour type ϕ. Dans notre exemple $\phi \equiv \tau_1 \rightarrow\ \cdots \rightarrow \tau_n \rightarrow \tau$. L'opérateur `fix` a pour type $(\phi \rightarrow \phi) \rightarrow \phi$ pour n'importe quel type ϕ. De même des opérateurs comme la conditionnelle `if/then/else`, le produit peuvent se traiter comme des constantes du point de vue du typage.

Règles de typage Les règles de typage de ce langage s'expriment sous la forme $\Gamma \vdash e : \tau$ dans lequel e est une expression, τ un type et Γ un environnement qui contient des variables associées à des types : $x_1 : \tau_1, \ldots, x_n : \tau_n$. On suppose que chaque constante primitive c est associée à un type $\tau(c)$. Il y a une règle de typage par construction d'expression :

$$\frac{}{\Gamma \vdash c : \tau(c)} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash f : \sigma \rightarrow \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash (f\ e) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fun}\ x \rightarrow e : \sigma \rightarrow \tau}$$

Inférence On se donne maintenant un ensemble de variables visibles et un terme e et on veut savoir s'il existe des types $\tau_1, \ldots, \tau_n$ et τ tels que $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau$.

Pour cela on étend le langage de type avec des variables de type (α), on parle alors de schéma de types. Le système de type s'étend naturellement si on remplace les types par des schémas de type. On peut associer des schémas de type plutôt que des types aux constantes. La règle de typage des constantes devient alors

$$\overline{\Gamma \vdash c : \rho(\tau(c))}$$

si $\tau(c)$ est le schéma de type associé à τ et si ρ est une substitution. Ainsi la conditionnelle `if` a pour schéma de type $\texttt{bool} \to \alpha \to \alpha \to \alpha$. Les jugements de typage suivants sont valides :

$$\Gamma \vdash \texttt{if} : \texttt{bool} \to \texttt{int} \to \texttt{int} \to \texttt{int} \quad \texttt{if} : \texttt{bool} \to \texttt{bool} \to \texttt{bool} \to \texttt{bool}$$

On peut substituer un schéma de type à la variable α :

$$\texttt{if} : \texttt{bool} \to (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha)$$

On se donne $(n+1)$ schémas de type $\sigma_1, \dots, \sigma_n$ et σ et on va construire un algorithme qui va décider s'il existe une substitution ρ telle que : $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$.

On remarque que si $x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash e : \sigma$, alors le jugement est toujours vrai si on applique une substitution aux schémas de type : $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$.
Par exemple : $\vdash \texttt{fun}\ x \to x : \alpha \to \alpha$ mais aussi
$\vdash \texttt{fun}\ x \to x : \sigma \to \sigma$ pour n'importe quel schéma de type σ.

Lorsqu'une substitution ρ existe telle que $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$. alors il en existe une principale ρ_0, i.e. $x_1 : \rho_0(\sigma_1), \dots, x_n : \rho_0(\sigma_n) \vdash e : \rho_0(\sigma)$ et pour tout ρ tel que $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$, alors il existe une substitution ρ' telle que $\rho = \rho' \circ \rho_0$.

L'algorithme prend en argument le contexte, le terme e à typer et le schéma de types σ associé : $x_1, \sigma_1, \dots, x_n, \sigma_n, e, \sigma$, il échoue s'il n'existe pas de substitution ρ telle que $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$.
et sinon il renvoie la substitution principale qui vérifie ce jugement. L'algorithme procède par analyse de la structure de l'expression e.

- $e = c$ une constante primitive, alors $\Gamma \vdash c : \rho(\tau(c))$ pour toute substitution ρ. Donc $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$ si et seulement si $\rho(\sigma) = \rho(\tau(c))$. Le problème se ramène à trouver s'il existe une substitution ρ telle que $\rho(\sigma) = \rho(\tau(c))$ qui est un problème de filtrage lorsque $\tau(c)$ est un type sans variable et un problème d'unification en général si $\tau(c)$ est un schéma de type (par exemple pour `fix` ou `if/then/else`). Dans les deux cas l'unification nous dit s'il existe une solution et nous en donne une principale.
- $e = x$ une variable, alors $\Gamma \vdash x : \phi$ si et seulement si $x : \phi \in \Gamma$. Donc on a $x = x_i$ et $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e : \rho(\sigma)$ ssi $\rho(\sigma_i) = \rho(\sigma)$ donc si ρ unifie σ et σ_i.
- $e = (e_1\ e_2)$ alors la règle de typage est :

$$\frac{\Gamma \vdash e_1 : \phi \to \psi \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash (e_1\ e_2) : \psi}$$

  On introduit donc une nouvelle variable de type α et on cherche à trouver une substitution telle que $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e_1 : \rho(\alpha \to \sigma)$ et $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n) \vdash e_2 : \rho(\alpha)$. Pour cela on rappelle l'algorithme récursivement avec $x_1, \sigma_1, \dots, x_n : \sigma_n, e_1, \alpha \to \sigma$ pour trouver ρ_0 principal tel que $x_1 : \rho_0(\sigma_1), \dots, x_n : \rho_0(\sigma_n) \vdash e_1 : \rho_0(\alpha \to \sigma)$. On peut ensuite rappeler l'algorithme récursivement avec $x_1, \rho_0(\sigma_1), \dots, x_n : \rho_0(\sigma_n), e_2, \rho_0(\alpha)$ pour trouver ρ_1 principal tel que $x_1 : \rho_1(\rho_0(\sigma_1)), \dots, x_n : \rho_1(\rho_0(\sigma_n)) \vdash e_2 : \rho_1(\rho_0(\alpha))$. Comme on a aussi $x_1 : \rho_1(\rho_0(\sigma_1)), \dots, x_n : \rho_1(\rho_0(\sigma_n)) \vdash e_1 : \rho_1(\rho_0(\alpha)) \to \rho_1(\rho_0(\sigma))$. La substitution recherchée est $\rho_1 \circ \rho_0$.
- $e = \texttt{fun}\ x \to e$ alors la règle de typage est

$$\frac{\Gamma, x : \phi \vdash e : \psi}{\Gamma \vdash \texttt{fun}\ x \to\ e : \phi \to \psi}$$

  Donc il faut trouver une substitution ρ tel que $\rho(\sigma) = \phi \to \psi$ avec $x_1 : \rho(\sigma_1), \dots, x_n : \rho(\sigma_n), x : \phi \vdash e : \psi$. On introduit donc deux nouvelles variables α et β et on appelle l'algorithme récursivement avec $x_1, \sigma_1, \ldots, x_n : \sigma_n, x : \alpha, e, \beta$ pour trouver ρ_0 tel que $x_1 : \rho_0(\sigma_1), \ldots, x_n : \rho_0(\sigma_n), x : \rho_0(\alpha) \vdash e : \rho_0(\beta)$. Il suffit ensuite de trouver ρ tel que $\rho_1(\sigma) = \rho_1(\rho_0(\beta))$. Si ρ_1 est l'unificateur principal de σ et $\rho_0(\beta)$ alors la substitution $\rho_1 \circ \rho_0$ répond à notre problème.

Exemples Montrer que l'expression `fun` x→(x x) n'est pas typable.

Polymorphisme à la ML L'algorithme précédent fournit pour chaque expression un schéma de type principal. Lorsque l'on introduit une nouvelle déclaration `let` $x = e$, l'algorithme, s'il réussit va trouver un schéma de type principal σ pour e. Il y a donc plusieurs types possibles τ pour x (toutes les instances de σ), et on peut à chaque fois introduire dans l'environnement $x : \tau$. Pour ne pas avoir à choisir, a priori, on peut introduire $x : \sigma$ dans l'environnement et on souhaite étendre la règle de typage en :

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \rho(\sigma)}$$

Ainsi si on introduit `let` id = `fun` $x \ \to\ x$, on trouve $\alpha \to \alpha$ comme schéma de type et on peut par exemple typer le terme $(id\ id)$.
Cependant l'ajout en toute généralité d'une règle

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \rho(\sigma)}$$

modifie profondément le système de type, ainsi on a :

$$\frac{x : \gamma \vdash x : \beta \to \alpha \quad x : \gamma \vdash x : \beta}{\vdash \texttt{fun}\ x \to (x\ x) : \alpha}$$

Ce système est connu sous le nom de lambda-calcul du second ordre ou encore lambda-calcul polymorphe ou encore système F. Il a été montré que l'inférence de type dans ce système n'était pas décidable.

Les langages ML adoptent un polymorphisme limité. Seules les variables qui apparaissent dans les schémas de types des variables déclarées dans des `let` peuvent être généralisées. De plus si la liaison `let` apparaît sous une abstraction `fun`, alors seules les variables qui n'apparaissent pas dans les schémas de type des paramètres peuvent être généralisées.

5.2.8 Autres analyses sémantiques

Le typage n'est qu'un exemple particulier d'analyse sémantique. Il existe d'autres techniques d'analyse.

Vérification de bonne formation des programmes

Les langages de programmation demandent parfois au programme de satisfaire certaines contraintes. Dans un programme Java, un champ marque `final` devra pouvoir être calculé statiquement, les échappements `break` ou `continue` associés à des labels doivent se trouver sous des instructions de même label, les cas d'une instruction `switch` doivent être calculables à la compilation.

Analyse de flots

L'analyse de flots s'intéresse à l'enchaînement de l'exécution des instructions d'un programme. Celui-ci est représenté par un graphe. À chaque point du graphe on calcule certaines informations relatives au programme. Ces informations sont propagées le long des arêtes du programme. Un calcul de point fixe permet de gérer les boucles dans l'exécution. L'analyse de flots se fait en général sur des langages de bas niveau.

Cette analyse dont un cas particulier sera étudié en TD permet par exemple de détecter du "code mort" c'est à dire des morceaux de code qui ne seront jamais exécutés ou bien de déterminer la durée de vie des variables (point de programme à partir duquel la variable n'est plus accédée et donc l'emplacement mémoire peut être récupéré). On peut également à l'aide d'analyse de flots calculer les alias possibles d'un programme, c'est-à-dire les variables qui peuvent pointer sur la même case mémoire (en cas de passage de variables par référence, de manipulation de pointeurs, de langage objet).

Interprétation abstraite

Cette technique consiste à calculer une approximation finie d'un programme. L'idée est de simuler l'exécution du programme sur des données *abstraites*, plus simples que les entrées du programme et qui vont garantir que l'exécution va pouvoir être faite dans un temps raisonnable. Une fois le calcul abstrait exécuté, on souhaite en déduire des informations sur le comportement du programme dans des cas concrets.

Des abstractions sont couramment utilisées en mathématique. Par exemple, on peut vérifier une suites d'opérations arithmétiques en effectuant un calcul modulo une certaine valeur. Si le calcul modulo ne satisfait pas l'équation alors il en est de même pour le calcul concret.

L'idée est d'avoir un ensemble A de données abstraites, un ensemble C de données concrètes et une relation entre ces données. Dans le calcul modulo p, A représentera les entiers modulo p et C les entiers. La relation entre $a \in A$ et $c \in C$ sera que $c \mathbf{mod}\ p = a$. On fait deux interprétations sémantiques du langage de programmation, l'une sur l'ensemble abstrait, l'autre sur l'ensemble concret de manière à ce que les deux interprétations restent en correspondance. Il faut également pouvoir effectuer de manière statique et si possible efficace le calcul de la valeur abstraite du programme, en particulier les calculs de point fixe.

Les techniques d'analyse de flots ou d'interprétation abstraite ne donnent que des résultats partiels sur le programme mais peuvent être mises en œuvre de manière automatique.

5.3 Attributs

Le calcul d'informations sémantiques sur les programmes se fait en utilisant la technique des grammaires attribuées. Celles-ci permettent non seulement d'analyser syntaxiquement une entrée mais également de calculer pour chaque nœud de l'arbre de dérivation syntaxique des informations qui sont spécifiées localement par une équation.

Les attributs sont des données associées à chaque symbole (terminal ou non terminal) d'une grammaire. Il est commode de voir ses données comme des objets étiquetés. On notera $x.l$ l'attribut d'étiquette l associé au symbole x.

5.3.1 Attributs synthétisés ou hérités

Définition 1 *Une définition syntaxique est constituée d'un ensemble de productions* $X := x_1 \dots x_k$ *associées à une suite de déclarations de la forme* $X.l := e$ *où* e *est une expression ne faisant intervenir que les attributs de* $x_1 \dots x_k$ *on dira alors que* $X.l$ *est un attribut synthétisé ou bien* $x_i.l_i = e$ *où* e *est une expression ne faisant intervenir que les attributs de* X *et* x_j *pour* $j \neq i$ *auquel cas on dira que* $x_i.l_i$ *est un attribut hérité.*

On peut voir cette définition syntaxique comme un ensemble d'équations entre des variables $y.l$ associées à chaque nœud de l'arbre et chaque attribut.

Comme le même symbole de l'alphabet peut apparaître plusieurs fois dans la même production, pour différencier les nœuds de l'arbre, on introduit des indices dans les définitions.

L'ensemble des déclarations ne forme pas forcément une définition valide des attributs de chaque symbole en effet rien n'empêche d'introduire des circularités.

Quelques restrictions On considère que les attributs des symboles terminaux ne peuvent être que synthétisés, ils sont en général produits directement par l'analyseur lexical. Par exemple un terminal correspondant au léxème `NUM` aura pour attribut la valeur de l'entier, un terminal correspondant au léxème `ID` pourra avoir comme attribut un index permettant de localiser l'identificateur dans la table des symboles.

On considère également que les attributs du symbole de départ ne peuvent être que synthétisés (c'est forcément le cas si ce symbole n'apparaît que dans la partie gauche d'une seule règle).

Exemple 1 (Attributs synthétisés) *L'exemple typique d'attribut synthétisé est la valeur d'une expression arithmétique :*

$$\begin{array}{lll}
S & := E & \{S.val := E.val\} \\
E & := E_1 + E_2 & \{E.val := E_1.val + E_2.val\} \\
E & := E_1 - E_2 & \{E.val := E_1.val - E_2.val\} \\
E & := \mathtt{NUM} & \{E.val := \mathtt{NUM}.val\}
\end{array}$$

Exemple 2 (Attributs hérités) *Un exemple d'attribut hérité est la déclaration d'identificateurs typés. On veut reconnaître les expressions* `type` $x_1, \dots, x_n$ *avec* `type` *un non-terminal correspondant à* `int` *ou* `real` *et déclarer les identificateurs du bon type dans la table des symboles par l'action* $\mathtt{add}(x, \mathtt{type}, T)$ *qui rend la table des symboles* T *enrichie par l'introduction de* x *de type* `type`*. L'attribut correspondant est noté* $L.a$*.*

$$\begin{array}{lll}
D & := T\ L & \{L.typ := T.typ\} \\
T & := \mathtt{int} & \{T.typ := \mathtt{int}\} \\
T & := \mathtt{real} & \{T.typ := \mathtt{real}\} \\
L & := L_1, id & \{L_1.typ := L.typ; L.a := \mathtt{add}(id.val, L.typ, L_1.a)\} \\
L & := id & \{L.a := \mathtt{add}(id.val, L.typ, \mathtt{vide})\}
\end{array}$$

Graphe de dépendance

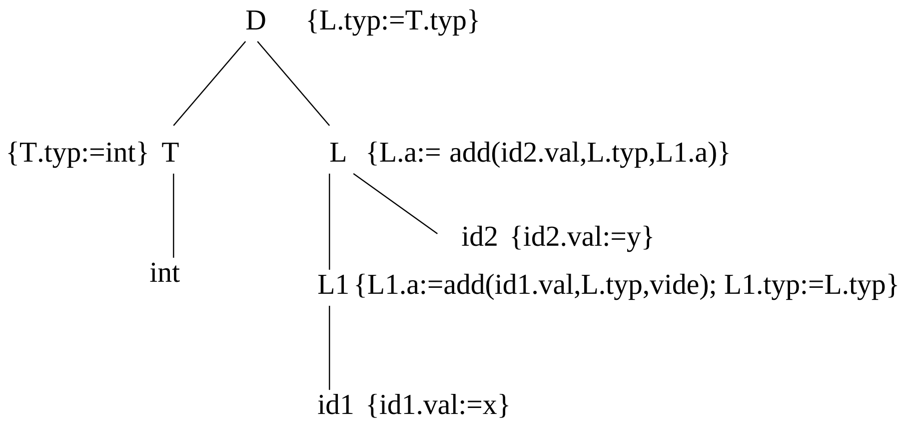
Lorsqu'on a analysé une expression, il est possible d'étiqueter les nœuds de l'arbre d'analyse syntaxique par les déclarations. Sur cet arbre syntaxique il est possible de construire un *graphe de dépendance* entre les attributs apparaissant dans chaque nœud. On aura une arête de $x.l$ à $y.m$ si le calcul de $y.m$ nécessite le calcul préalable de $x.l$ c'est à dire s'il y a une déclaration $y.m := f(\dots, x.l, \dots)$.

Dans le cas d'attributs synthétisés, les arêtes sont dirigées des fils vers le père. Dans la cas d'attributs hérités on peut avoir une arête du père vers le fils ou d'un frère vers un autre.

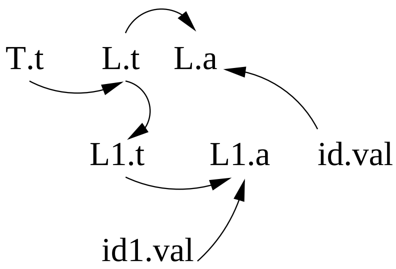
Par exemple si on doit reconnaître l'expression

$$\texttt{int}\ x, y$$

Alors l'arbre de syntaxe étiqueté par les productions est :



Le graphe de dépendance est :



Le graphe de dépendance définit une relation sur les nœuds de l'arbre associés à des étiquettes. Si cette relation peut être complétée en un ordre strict total alors on peut numéroter les attributs dans les nœuds de l'arbre d'analyse syntaxique dans un ordre permettant d'effectuer les calculs.

Dans l'exemple, une possibilité d'ordre d'évaluation est :

$$T.t < L.t < id.val < L.a < L1.t < id1.val < L1.a$$

Différents modes d'évaluation

On peut envisager différents modes pour l'évaluation des attributs.

- Le plus général consiste à construite au moment de l'analyse en même temps que l'arbre de syntaxe abstraite, le graphe de dépendance pour lequel on cherchera un ordre d'évaluation.
- On peut déterminer pour chaque production dans quel ordre l'évaluation des attributs doit se faire.
- On peut avoir un ordre prédéterminé d'évaluation quelle que soit la forme des productions.

5.3.2 Évaluation des S-attributs

Définition 2 (Définition S-attribuée) *Une définition syntaxique est dite S-attribuée si elle ne comporte que des attributs synthétisés.*

Il est alors très simple d'évaluer ces attributs lors d'une phase d'analyse ascendante.

Lors de l'analyse ascendante, la pile comporte en théorie une suite $s_0 x_1 s_1 \dots x_n s_n$ formée d'états et de symboles. Chaque état sauf l'état initial est couplé avec un symbole et on pourrait donc représenter la pile comme

s_n $\vdots$	x_n $\vdots$
s_1	x_1
s_0	

La présence des symboles n'est pas utile dans l'algorithme de reconnaissance mais sert à la compréhension. On ajoute à la pile un champ pour stocker les attributs associés au symbole :

s_n $\vdots$	x_n $\vdots$	$x_n.l_n = p_n$
s_1	x_1	$x_1.l_1 = p_1$
s_0		

Lorsqu'une lecture est faite, on empile le terminal et ses attributs fournis par l'analyseur lexical. Lorsqu'une réduction est faite, elle correspond à une production dont la partie droite est au sommet de la pile qui a donc la forme :X::= $x_k \dots x_n$. Les déclarations associées d'attributs synthétisés sont de la forme : $X.l := f(x_k.l_k, \dots, x_n.l_n)$. Il suffit donc au moment de la réduction de calculer le nouvel attribut associé au non-terminal qui arrive au somment de la pile :

s	X	$X.l = f(x_k.l_k, \dots, x_n.l_n)$
s_{k-1} $\vdots$	x_{k-1} $\vdots$	$x_{k-1}.l_{k-1} = p_{k-1}$
s_1	x_1	$x_1.l_1 = p_1$
s_0		

5.3.3 Évaluation des L-attributs

Les attributs hérités peuvent poser des problèmes à l'évaluation. On va s'intéresser à une classe particulière d'attributs hérités qui s'appelle les $L-attributs$.

Définition 3 (Définition L-attribuée) *Une définition syntaxique est L-attribuée si dans les déclarations associées à une production X::= $x_1 \dots x_n$, les attributs hérités de x_i ne dépendent que des attributs hérités de X et des attributs hérités ou synthétisés de x_j pour $j < i$.*

Toute définition S-attribuée est aussi L-attribuée puisque la restriction ne concerne que les attributs hérités.

Notre définition des déclarations de type est L-attribuée.

Schéma de traduction

Lorsqu'une définition est L-attribuée il est possible de l'écrire sous une forme un peu différente.

Si on a une production $X := x_1 \ldots x_n$, on insère les déclarations des attributs hérités de x_i juste avant l'occurrence de x_i. Les déclarations des attributs synthétisés de X restent à la fin de la production.

Dans le cas de notre exemple on obtient :

$$\begin{array}{lll}
D & := T\ \{L.t := T.t\}\ L & \\
T & := \texttt{int} & \{T.t := \texttt{int}\} \\
T & := \texttt{real} & \{T.t := \texttt{real}\} \\
L & := \{L_1.typ := L.typ\}\ L_1, id & \{L.a := \texttt{add}(id.val, L.typ)\} \\
L & := id & \{L.a := \texttt{add}id.val, L.typ\}
\end{array}$$

Analyse descendante de L-attributs

Le calcul de L-attributs se fait naturellement dans une phase d'analyse descendante. L'idée est que au moment de chercher à analyser un terminal X, ces attributs hérités sont connus et lorsqu'on aura fini de reconnaître la suite de terminaux correspondant à ce non-terminal les attributs synthétisés seront complètement calculés.

Cela correspond à une évaluation des attributs par un parcours en profondeur de l'arbre de syntaxe.

Il est très simple d'écrire un tel analyseur de manière fonctionnelle, à chaque symbole est associé une fonction qui attend comme argument les attributs hérités et renvoie après reconnaissance les attributs synthétisés.

Si on a une règle $X := x_1 \ldots x_n$ avec des attributs h hérités et s synthétisés, on a $X.s := f(X.h, x_1.h, x_1.s, ..., x_n.h, x_n.s)$ et $x_i.h := h_i(X.h, x_1.h, x_1.s, \ldots, x_{i-1}.h, x_{i-1}.s)$. On lui fait correspondre une fonction :

```
let X(X.h) = let x_1.h = h_1(X.h)
          in let x_1.s = x_1(x_1.h)
          in let x_2.h = h_2(X.h,x_1.h,x_1.s)
          in let x_2.s = x_2(x_2.h)
          in ..
          in let x_n.h =  h_n(X.h,x_1.h,x_1.s,..x_(n-1).h,x_(n-1).s)
          in let x_n.s = x_n(x_(n-1).h)
          in f(X.h,x_1.h,x_1.s,...,x_n.h,x_n.s)
```

La difficulté est que si on veut reconnaître à la fois l'expression et calculer les attributs, il est nécessaire que la grammaire soit LL. On a vu comment transformer des grammaires pour par exemple éliminer la récursion gauche. Il faut également être capable de transformer le calcul d'attributs de manière correcte.

Chapitre 6

Génération de code intermédiaire

6.1 Introduction

Avant d'engendrer un code particulier, nous allons voir les différents problèmes qui se posent pour organiser les données et les instructions dans un programme.

6.1.1 Modèle d'exécution

Lors de l'exécution du code, une partie de la mémoire est réservée aux instructions du programme avec un pointeur pour indiquer où on est dans l'exécution. La taille de ce code est connue à la compilation. On suppose que l'on a un contrôle séquentiel de l'exécution d'un programme. C'est-à-dire que sauf instruction explicite de saut, les instructions du programmes sont exécutées l'une après l'autre.

Les données du programme sont stockées soit dans la mémoire soit dans les registres de la machine. La mémoire est organisée en mots, on y accède par une adresse qui est représentée par un entier. Les valeurs simples sont stockées dans une unité de la mémoire, les valeurs complexes (tableaux, structures) sont stockées dans des cases consécutives de la mémoire, ou dans des structures chaînées (une liste pourra être représentée par la valeur de son premier élément associée à l'adresse du reste de la liste).

Les registres permettent d'accéder rapidement à des données simples. Le nombre de registres dépend de l'architecture de la machine et est en général limité.

6.1.2 Allocation

Certaines données sont explicitement manipulées par le programme source par l'intermédiaire de variables, d'autres sont créées par le compilateur par exemple pour conserver le résultat de valeurs intermédiaires lors de l'exécution d'un calcul arithmétique, ou pour conserver des environnements associés à des valeurs fonctionnelles.

Certaines données ont une durée de vie connue à la compilation, il est donc possible et souhaitable de réutiliser la place occupée. L'analyse de la durée de vie des variables doit au moins tenir compte des règles de portées du langage (une variable qui n'est plus visible correspond en général à une donnée qui n'est plus accessible), elle peut également utiliser des techniques d'analyse de flots qui permettent de détecter qu'une variable n'est pas utilisée.

Les variables globales du programme peuvent être allouées à des adresses fixes par le compilateur, à condition de connaître leur taille.

La gestion des variables locales des blocs et des procédures, ou bien le stockage de valeurs intermédiaires se prête bien à une gestion de l'allocation de la mémoire en pile.

D'autres données ont par contre une durée de vie qui n'est pas connue à la compilation. C'est le cas lorsque l'on manipule des pointeurs. Certains objets alloués ne correspondront pas directement à une valeur mais seront accessibles par l'intermédiaire de leur adresse qui pourra être associée à une variable. Ces données vont être allouées en général dans une autre partie de la mémoire, organisée en tas (heap en anglais). Pour ne pas saturer cet espace il sera nécessaire soit que le programmeur utilise des commandes explicites pour libérer l'espace, soit que l'exécution du programme utilise une programme général de récupération de mémoire appelé communément *gc* pour *garbage collector* qui est traduit en *glaneur de cellules* ou *ramasse-miettes*.

6.1.3 Variables

Une variable est un identificateur apparaissant dans le programme qui est associé à un objet manipulé par le programme, en général stocké en mémoire. On distingue la *valeur gauche* de la variable qui représente l'adresse mémoire à partir de laquelle est stockée l'objet et la *valeur droite* de la variable qui représente la valeur de l'objet. Certains langages comme ML distinguent explicitement les deux objets, dans Pascal c'est la position syntaxique de la variable qui détermine si celle-ci doit être interprétée comme une valeur gauche ou droite. Le passage de paramètres aux procédures peut se faire en transmettant les valeurs gauches ou les valeurs droites des objets manipulés.

Environnement et état On appelle *environnement* la liaison de noms à des adresses mémoires. Cette liaison peut être représentée par un ensemble fini de couples formés d'un identificateur et d'une adresse. Un environnement définit une fonction partielle qui associe une adresse à un identificateur.

On appelle *état* la liaison d'adresses à des valeurs qui est une fonction partielle des adresses dans les valeurs.

Lors d'une affectation d'une valeur à une variable, seule l'état change. Lors de l'appel d'une procédure comportant des paramètres ou des variables locales, l'environnement change. La variable introduite correspond à une nouvelle liaison entre un nom et une adresse.

6.1.4 Sémantique

Afin de construire une traduction correcte, il est essentiel de connaître les règles d'évaluation du langage source, c'est-à-dire sa sémantique. Ces règles sont souvent données en terme de transformation de l'environnement et/ou de l'état.

6.1.5 Procédures et fonctions

Une procédure a un nom, des paramètres, une partie de déclarations de variables ou de procédures locales et un corps. Une fonction est une procédure qui renvoie une valeur.

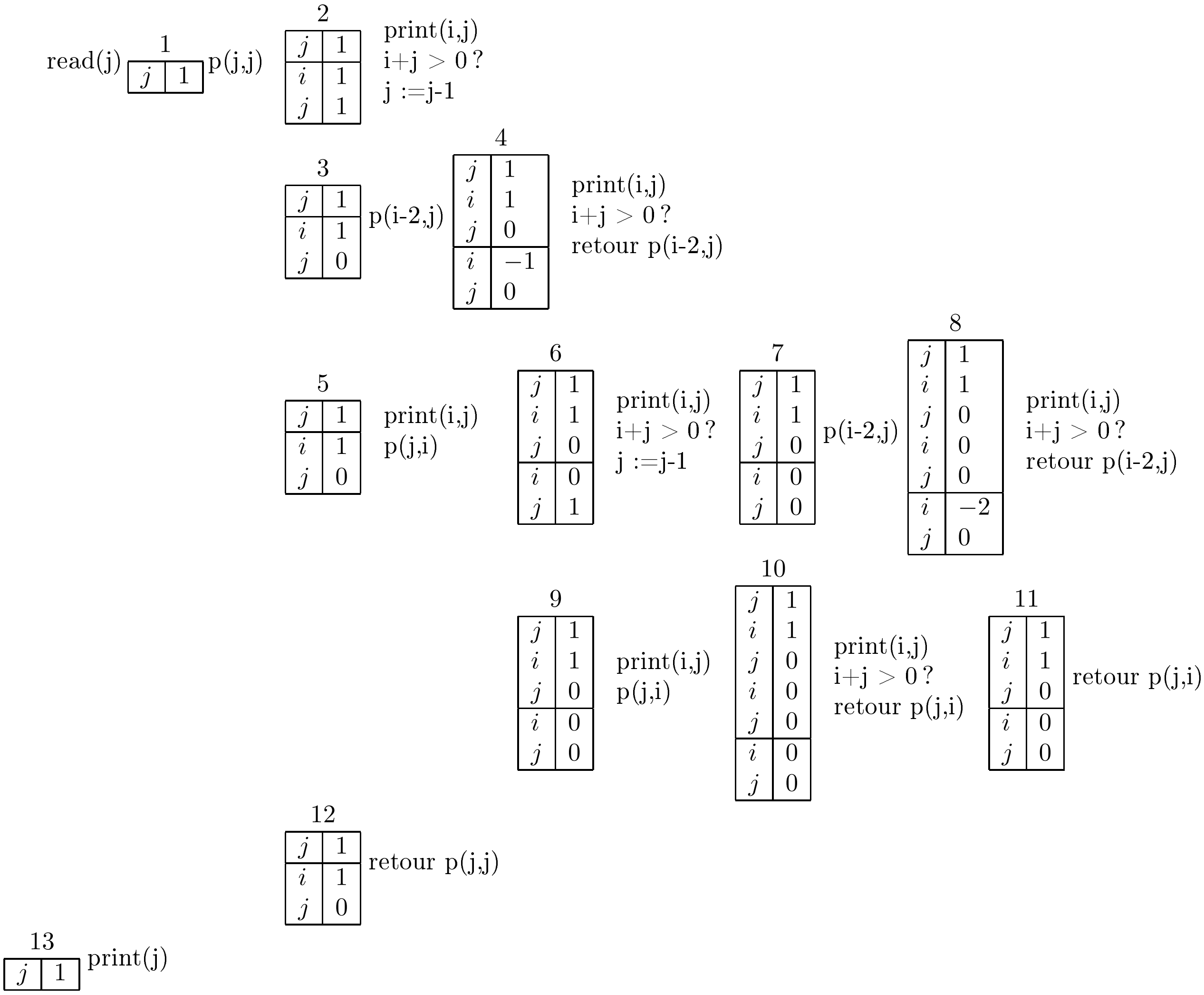
Les *paramètres formels* sont des variables locales à la procédures qui seront instanciées lors de l'appel de la procédure par les *paramètres effectifs*. La procédure peut déclarer des variables locales qui seront initialisées dans le corps de la procédure. L'espace mémoire alloué pour les paramètres et les variables locales peut en général être libéré lorsque l'on sort de la procédure. Cependant on ne peut pas contrôler en général le nombre d'appel d'une procédure ni le moment où celle-ci sera appelée, il n'est donc pas possible de donner statiquement les adresses des variables apparaissant dans la procédure. Ces adresses pourront être calculées relativement à l'état de la pile au moment de l'appel de la procédure.

Exemple

```
program main
var j : int;
procedure p(i,j:int);
{print(i,j);if i+j>0 then j:=j-1; p(i-2,j); print(i,j);p(j,i)}
{read(j); p(j,j); print(j)}
```

Le nombre d'exécutions de p dépend de la valeur lue en entrée. À chaque entrée dans la procédure p deux nouvelles variables i et j sont allouées.

Dans le schéma suivant, on a représenté l'exécution de ce programme pour une valeur lue de 1. Les objets dans la pile sont représentés après chaque instruction qui en modifie l'état. La pile est représentée en grossissant vers le bas, les valeurs courantes de i et de j étant séparées. L'exécution se lit de gauche à droite et de bas en haut, les piles sont numérotées dans l'ordre de l'exécution.



6.1.6 Organisation de l'environnement

L'organisation de l'environnement d'exécution dépend de ce que le langage autorise :
- procédures récursives
- traitement des variables locales
- référence à des noms non locaux
- mode de passage des paramètres
- existence de données de taille variable

- possibilité de renvoyer une procédure ou de prendre une procédure en argument
- allocation/desallocation dynamique de données par le programmeur

6.2 Tableau d'activation

Un des points importants de l'organisation de la mémoire est la gestion de l'appel de procédure (ou de fonction). Celui-ci se fait par l'intermédiaire d'un *tableau d'activation*. Le tableau d'activation est une portion de la mémoire qui est allouée à l'appel de la procédure et restaurée à la sortie. Il contient toutes les informations nécessaires à l'exécution du programme et sauvegarde les données qui devront être restaurées à la sortie de la procédure.

Afin de faciliter la communication entre des codes compilés à partir de langages distincts (par exemple pour appeler des procédures de bas niveau à partir d'un langage de haut niveau), il n'est pas rare qu'un format de tableau d'activation pour une architecture donnée soit particulièrement recommandé.

6.2.1 Données à conserver

Lors d'un appel de procédure le contrôle du code est modifié, dans le cas d'un retour normal de la procédure (pas d'échappement en erreur ou d'instruction de type `goto`) la suite de l'exécution doit se poursuivre à l'instruction suivant l'instruction de l'appel. Le compteur d'instruction doit donc être sauvegardé à chaque appel de procédure.

On a vu que les données locales à la procédure s'organisaient dans la pile à partir d'une adresse qui était déterminée à l'exécution et en général sauvée dans un registre. Lorsqu'une nouvelle procédure est appelée, cette valeur change, il est donc nécessaire de sauvegarder la valeur courante qui pourra être restaurée à la fin de la procédure.

6.2.2 Passage de paramètres

Les paramètres formels de la procédure sont des variables qui sont initialisées lors de l'appel de la procédure. Il y a plusieurs manières d'effectuer cette initialisation. On suppose que l'on a une procédure p avec un paramètre formel x qui est appelée avec le paramètre effectif e. On examine différents mode de passage de ce paramètre.

Passage par valeur

Dans le passage de paramètre par valeur, x est une nouvelle variable allouée localement par la procédure dont la valeur est le résultat de l'évaluation de e. Après la fin de la procédure les modifications apportées à x ne sont plus visibles. En l'absence de pointeurs, les seules variables modifiées sont les variables non locales à la procédure explicitement nommées dans les instructions du programme. Il est nécessaire de réserver une place proportionnelle à la taille du paramètre ce qui peut être coûteux dans le cas de tableaux.

Passage par référence ou par adresse

On calcule la valeur gauche de l'expression e (si e n'a pas de valeur gauche on crée une variable que l'on initialise à la valeur droite de e et on utilise la valeur gauche de cette variable). La procédure alloue une variable x qui est initialisée par la valeur gauche de e. Toute référence à x dans le corps de la procédure est interprétée comme une opération sur l'objet situé à l'adresse stockée en x. Ce mode de passage occupe une place indépendante de la taille du paramètre.

Passage par nom

Il s'agit d'un remplacement textuel dans le corps de la procédure des paramètres formels par les paramètres effectifs. C'est un mécanisme de macro qui peut provoquer des problèmes de capture.

Exemple

```
swap(x,y:int)
var z:int
{z:=x; x:=y; y:=z}
```

Si z et t sont des variables globales du programme `swap(z,t)` n'aura pas le comportement attendu.

On peut renommer les variables locales de manière à ne pas interagir avec les variables apparaissant dans les arguments. En effet le nom de la variable locale z, tant qu'il est différent de x et y n'a pas d'influence sur le code de la procédure.

Cela ne suffit pas par exemple l'appel par nom de `swap(i,a[i])` ne donne pas le résultat attendu.

Cependant cette méthode peut être utile lors de la compilation de procédures de petite taille où le coût de la gestion de l'appel est important.

Passage par copy-restore

Lors de l'appel de la procédure la valeur droite de e sert à initialiser la variable x. À la sortie de la procédure la valeur droite de x sert à mettre à jour la valeur gauche de e.

Cette méthode peut avoir, dans des cas particuliers, des comportements différents du passage par référence comme le montre l'exemple suivant.

```
program main;
var a : integer;
procedure p(x:integer);
  {x:=2; a:=0}
{a:=1;p(a);write(a)}
```

Évaluation paresseuse Dans les langages fonctionnels, on distingue les langages *stricts* des langages dits *paresseux*. Dans un langage strict, les valeurs des arguments sont évalués avant d'être passées en paramètre à une fonction. C'est le cas des langages CAML ou SML. Dans les langages paresseux, l'expression passée en argument à une fonction f sera évaluée seulement si f a besoin de cette valeur (on parle d'appel par nécessité). Supposons que la fonction $f(x) = t$ soit déclarée et que l'on veuille calculer $f(e)$ alors dès que l'évaluation de t nécessitera la valeur de x, le calcul de e sera effectué. Si t utilise plusieurs fois x, le calcul sera effectué une seule fois, l'expression e passée en argument vient avec son environnement (structure représentant les valeurs des variables utilisées par e) ce qui évite les problèmes de capture. ce mécanisme est donc différent de celui de remplacement textuel utilisé dans les macros.

Le langage Haskell, utilise l'évaluation paresseuse. Celle-ci a l'avantage de n'exécuter que les calculs utiles. cependant le fait de devoir passer en paramètre un code à exécuter comportant un environnement d'évaluation induit un surcoût. De plus dans ces langages, il devient difficile de prédire le moment ou des effets de bord inclus dans les expressions auront lieu. Aussi Haskell est un langage fonctionnel pur sans effet de bord. ce qui ne l'empêche pas d'être efficace, diffusé de manière industrielle et utilisé assez largement.

6.2.3 Accès aux variables locales

Une procédure peut accéder à ses variables locales qui se trouvent dans son tableau d'activation et aux variables globales qui sont à une adresse connue à la compilation. Dans les langages fonctionnels ou Pascal, les procédures peuvent être imbriquées et le corps d'une procédure p peut accéder à des variables déclarées dans une procédure englobante q. Il faudra donc accéder au tableau d'activation de la procédure q qui fort heureusement se trouve alloué sur la pile. Malheureusement la position du tableau d'activation de q par rapport à p ne peut être déterminée statiquement. Il faut donc utiliser un mécanisme de chaînage des tableaux d'activation pour retrouver la bonne variable.

Arbre de niveau des déclarations On suppose que l'on a un langage fonctionnel ou impératifs dans lequel procédures, fonctions et variables peuvent être arbitrairement imbriquées. La portée des variables se fait de manière statique suivant les règles habituelles de visibilité.

Le niveau d'une déclaration (procédure, fonction ou variable) est le nombre de procédures ou de fonctions sous lesquelles elle est déclarée). Le programme principal a un niveau 0, les variables globales définies dans le programme principal auront le niveau 1.

Exemple

```
program main
var t:int
procedure p(x,y:int);
  var z : int
  procedure q(i:int)
  var u : int
    begin u:=z+t+i;
    if x=u or y=u then x
    else if u < x then p(u,x)
    else if y < u then p(y,u)
    else q(i+1)
    end
  begin read(z); q(z) end
begin read(t);p(t,t) end
```

On peut calculer de manière statique les niveaux de chaque identificateur.

On introduit un certain nombre de définitions concernant les déclarations, un identificateur désigne indifféremment une procédure, une fonction ou une variable : On dira que p est le *père* d'un identificateur y si y est déclaré dans p. On dira que p est un *ancêtre* d'un identificateur y si p est soit y soit le père d'un ancêtre de y. On dira que deux identificateurs sont *frères* s'ils ont le même père.

Si le corps d'une procédure p mentionne un identificateur alors les règles de portées sont telles que celui-ci est soit une déclaration locale de p soit un ancêtre de p (par exemple p lui-même) soit un frère d'un ancêtre de p.

Arbre d'activation On s'intéresse maintenant aux exécutions possibles d'un programme, pour cela on introduit la notion d'arbre d'activation.

Les nœuds de cet arbre représentent des appels de procédures $p(e_1, \ldots, e_n)$. Un nœud a k fils $q_1, \ldots q_k$ si l'exécution du corps de la procédure donne lieu directement aux appels de procédures $q_1, \ldots q_k$ (ces appels pouvant eux-mêmes engendrer de nouveaux appels de procédures). La racine de l'arbre est l'exécution du programme principal.

Exemple Donner les arbres d'activation du programme introduit précédemment dans les cas où les variables lues valent successivement 1, 1, et 0.

Lors d'une exécution du code on dira qu'une procédure est active si on n'a pas encore fini d'exécuter le corps de cette procédure.

On remarque que l'ordre des appels des procédures ne correspond pas à l'ordre de déclaration. Cependant on peut montrer simplement que lorsque qu'une procédure p est active alors tous les ancêtres de p sont des procédures actives. On fait une récurrence sur la profondeur de l'appel de p dans l'arbre d'activation. En effet c'est le cas initialement, le programme principal n'ayant que lui comme ancêtre. Soit maintenant une procédure active p, elle a été activée par une procédure q qui est toujours active et dont les ancêtres (hypothèse de récurrence) sont actifs. Maintenant pour des raisons de visibilité, on sait que soit q est le père de p soit p est le frère d'un ancêtre de q. Dans les deux cas tous les ancêtres de p sont bien actifs.

Toutes les procédures actives ont leur tableau d'activation réservé en mémoire qui contient leurs variables locales. Si une procédure active mentionne une variable y celle-ci a été déclarée localement par un de ses ancêtres (peut-être lui-même). Le degré de parenté de cet ancêtre est connu à la compilation. En chaînant chaque tableau d'activation d'une procédure p avec le tableau d'activation de son père, on retrouve simplement en suivant le bon nombre d'indirection l'endroit où la variable concernée est stockée.

Exemple Dans l'exemple donné précédemment le programme principal peut appeler p mais pas q, p peut appeler q et q peut appeler p.

6.2.4 Organisation du tableau d'activation

Si on suppose que l'adresse du bloc d'activation locale est donnée par le registre *fp* et le compteur d'instruction est donné par le registre *pc* alors une organisation possible du tableau d'activation est la suivante.

	valeurs intermédiaires ⋮	
	var. loc. k	*fp*+k-1
	⋮	} Variables locales
fp →	var. loc. 1	*fp*+0
	fp de la procédure père	*fp*-1
	fp de la procédure appelante	*fp*-2
	pc au moment de l'appel	*fp*-3
	param n	*fp*-4
	⋮	} Arguments
	param 1	*fp*-n-3
	val. retour	*fp*-n-4
	⋮	

En fait certaines de ses valeurs comme la valeur de retour (dans le cas de fonctions), ou même les paramètres sont souvent passées dans des registres.

Passage de paramètres de taille variable Certains langages comme Pascal autorisent le passage par valeur d'arguments de type tableau dont la taille est également un paramètre de la procédure et n'est donc pas connu à la compilation. Pour compiler le corps de la procédure, on décide de stocker à un endroit donné l'adresse du tableau. Les données de taille variable peuvent ensuite être alloués au sommet de la pile.

Appelant-Appelé Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé. Le code géré par l'appelant doit être écrit pour chaque appel tandis que celui écrit dans l'appelé n'apparaît qu'une seule fois.

L'appelant effectue la réservation pour la valeur de retour dans le cas de fonctions et évalue les paramètres effectifs de la procédure. Il peut se charger également de la sauvegarde des registres courants (compteur d'instruction, adresse du bloc d'activation) et calcule l'adresse du bloc d'activation père de la procédure appelée.

L'appelé initialise ses données locales et commence l'exécution. Au moment du retour, l'appelé place éventuellement le résultat de l'évaluation à l'endroit réservé par l'appelant et restaure les registres.

6.3 Code intermédiaire

6.3.1 Introduction

On utilise souvent un code intermédiaire indépendant de la machine cible. Cela permet de factoriser une grande partie du travail de compilation et de rendre le compilateur plus simplement portable.

Le choix d'un langage intermédiaire est très important, il doit être assez riche pour permettre un codage aisé des opérations du langage source sans créer de trop longues séquences de code. Il doit également être assez limité pour que l'implantation finale ne soit pas trop coûteuse.

Nous allons décrire les opérations d'une machine à pile particulière puis nous expliquerons comment engendrer le code associé à des constructions de langage.

Notations Nous décrirons le langage considéré par des règles de grammaire.

Nous spécifierons une fonction *code* qui prend comme argument un arbre de syntaxe abstraite pour le langage et renvoie une suite d'instructions du code de la machine à pile. Par souci de lisibilité nous utilisons les notations concrètes pour représenter la syntaxe abstraite.

Par exemple si $E := E_1 + E_2$ est une règle de grammaire nous écrirons $code(E_1 + E_2) = \ldots code(E_1) \ldots code(E_2)$ pour spécifier la valeur de la fonction *code* sur un arbre de syntaxe abstraite correspondant à l'addition de deux expressions.

Si C_1 et C_2 représentent des suites d'instructions alors $C_1 \mid C_2$ représente la suite d'instructions obtenue en concaténant C_2 à la suite de C_1. La liste vide d'instructions est représentée par $[]$.

On précisera au fur et à mesure les informations qui seront nécessaires à la génération du code.

6.3.2 Les principes de base d'une machine à pile particulière

On a une machine qui contient une zone de code, C, un registre *pc* (pointeur de code) contient l'adresse de l'instruction suivante à exécuter. Les instructions ont un nom suivi éventuellement de 1 ou 2 arguments. Ces arguments seront en général des entiers ou bien des étiquettes symboliques indiquant des instructions du code qui seront traduites en des entiers lors d'une phase de pré-analyse.

La machine contient une pile P permettant de stocker des valeurs, un registre *sp* (stack pointer) pointe sur la première cellule libre de la pile. Un autre registre *gp* (global pointer) pointe sur la base de la pile, endroit où seront stockées les variables globales.

Cette pile peut contenir des entiers, des flottants ou des adresses. On conviendra que certaines données grosses en taille comme les chaînes de caractères sont allouées dans un espace supplémentaire, lorsqu'elle devra manipuler une chaîne, la pile se contentera de manipuler l'adresse de la chaîne.

On présente pour chaque instruction de la machine, les modifications apportées à l'état de la pile et aux différents registres. Par exemple :

Code	Pile	sp	pc	Condition
PUSHI n	$\mathsf{P}[sp]$≔n	sp+1	pc+1	n est une valeur entière

signifie que le langage intermédiaire a une commande **PUSHI** qui attend un argument n qui doit être un entier. Si cette exécution a lieu alors que la pile vaut P, que le registre de sommet de pile vaut sp et le compteur de programme vaut pc alors après l'exécution du programme, la pile aura été modifiée en donnant la valeur n à l'adresse sp. Les registres sp et pc auront augmenté de 1. Cette instruction ne s'exécute correctement que si les conditions sont vérifiées sinon il y a erreur d'exécution.

6.3.3 Expressions arithmétiques

La machine ne permet d'effectuer une opération arithmétique qu'entre les deux valeurs au sommet de la pile, ces deux valeurs sont retirées de la pile et remplacées par une seule valeur représentant le résultat de l'opération.

Code	Pile	sp	pc	Condition
ADD	$\mathsf{P}[sp-2]$≔$\mathsf{P}[sp-2] + \mathsf{P}[sp-1]$	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
SUB	$\mathsf{P}[sp-2]$≔$\mathsf{P}[sp-2] - \mathsf{P}[sp-1]$	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
MUL	$\mathsf{P}[sp-2]$≔$\mathsf{P}[sp-2] * \mathsf{P}[sp-1]$	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
DIV	$\mathsf{P}[sp-2]$≔$\mathsf{P}[sp-2]/\mathsf{P}[sp-1]$	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers, si diviseur nul alors erreur

On pourra avoir un jeu d'instructions analogues mais qui agissent sur des flottants plutôt que sur des entiers. **FADD,FSUB,FMUL,FDIV**

Comparaisons

Les valeurs booléennes pour le résultat des comparaisons peuvent être représentées par des entiers. L'entier 1 représentera la valeur *vrai* et 0 la valeur *faux*. L'égalité correspond à la même instruction, que l'on compare des entiers, des flottants ou des adresses.

Code	Pile	sp	pc	Condition
INF	$\mathsf{P}[sp-2]$≔ si $\mathsf{P}[sp-2] < \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
INFEQ	$\mathsf{P}[sp-2]$≔ si $\mathsf{P}[sp-2] \leq \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
SUP	$\mathsf{P}[sp-2]$≔ si $\mathsf{P}[sp-2] > \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
SUPEQ	$\mathsf{P}[sp-2]$≔ si $\mathsf{P}[sp-2] \geq \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers
EQUAL	$\mathsf{P}[sp-2]$≔ si $\mathsf{P}[sp-2] = \mathsf{P}[sp-1]$ alors 1 sinon 0	$sp-1$	pc+1	$\mathsf{P}[sp-2]$, $\mathsf{P}[sp-1]$ entiers, flottants, adresses

On pourra avoir un jeu d'instructions analogues mais qui agissent sur des flottants plutôt que sur des entiers. **FINF,FINFEQ,FSUP,FSUPEQ,PUSHS**

Génération de code

La représentation des expressions arithmétiques sous forme binaire permet d'engendrer simplement un code permettant le calcul des expressions arithmétiques.

L'invariant à préserver est que après l'exécution de $code(E)$ à partir d'un état où $sp = n$ alors les valeurs de $\mathsf{P}[m]$ pour $m < n$ n'ont pas été modifiées, la valeur de l'expression E est calculée dans $\mathsf{P}[n]$ et $sp = n + 1$.

On suppose que les expressions arithmétiques sont engendrées par la grammaire :

$$E ::= entier \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1/E_2$$

Le code correspondant est donné par :

$$\begin{array}{ll} code(entier) & = \textbf{PUSHI}\ entier \\ code(E_1 + E_2) & = code(E_1) \mid code(E_2) \mid \ \textbf{ADD} \\ code(E_1 - E_2) & = code(E_1) \mid code(E_2) \mid \ \textbf{SUB} \\ code(E_1 * E_2) & = code(E_1) \mid code(E_2) \mid \ \textbf{MUL} \\ code(E_1/E_2) & = code(E_1) \mid code(E_2) \mid \ \textbf{DIV} \end{array}$$

6.3.4 Variables

Variables globales

Si le langage permet de mémoriser certaines valeurs dans des variables alors il nous faudra dans la machine deux instructions supplémentaires, l'une pour affecter une valeur calculée dans la pile à un espace réservé pour les valeurs globales, l'autre pour empiler au sommet de la pile une valeur contenue dans un variable globale.

Si a est une adresse dans la pile et n un entier alors $a + n$ désigne l'adresse obtenue n emplacements au-dessus de a.

Ces deux instructions sont les suivantes :

Code	Pile	sp	pc	Condition
PUSHG n	$\mathsf{P}[sp] := \mathsf{P}[gp+n]$	$sp+1$	$pc+1$	n entier $gp+n<sp$
STOREG n	$\mathsf{P}[gp+n] := \mathsf{P}[sp-1]$	$sp-1$	$pc+1$	n entier $gp+n<sp$

L'entier associé à chaque variable *id* affectée sera calculé au moment de l'analyse syntaxique ou sémantique et associé à l'identificateur par exemple dans la table des symboles et noté *adr(id)*. Il ne faut pas que les calculs intermédiaires écrasent les valeurs stockées, il est donc important de réserver la place nécessaire pour toutes les variables globales avant de démarrer les premiers calculs, le pointeur de sommet de pile initialement devra pointer sur la première case libre suivant les places réservées pour les globaux. Pour cela on utilisera une commande qui stocke n valeurs 0 dans la pile. On introduit également l'instruction duale qui permet de faire reculer de n espaces le pointeur de pile ce qui revient à effacer n valeurs.

Code	Pile	sp	pc	Condition
PUSHN n	$\mathsf{P}[sp+i] := 0 \quad \forall i.sp \leq i < sp+n$	$sp+n$	$pc+1$	n entier
POPN n		$sp-n$	$pc+1$	n entier

On suppose que notre grammaire des expressions est enrichie pour accéder à des variables globales représentant des entiers.

$$E ::= id$$

Le code correspondant est défini par

$$code(id) = \textbf{PUSHG}\ adr(id)$$

Dans le cas d'une variable de taille 1. Si la variable représente une donnée stockée sur k mots, il faudra exécuter :

$$code(id) = \textbf{PUSHG}\ adr(id)\textbf{PUSHG}\ adr(id)+1\ldots\textbf{PUSHG}\ adr(id)+(k-1)$$

On considère maintenant un langage dont les seules instructions sont des suites d'affectations.

L'analyse sémantique doit déterminer pour chaque variable globale *id* une place (relative par rapport à *gp*) notée *adr(id)* où la valeur correspondante sera stockée.

On ajoute un non-terminal à la grammaire pour représenter les instructions du langage. On étend la fonction *code* aux instructions.

On a les règles de grammaire :

$$\begin{array}{ll} I & ::= \epsilon \\ I & ::= I_1\ A; \\ A & ::= id{:=}E \end{array}$$

Et la génération de code correspondante :

$$\begin{array}{ll} code(\epsilon) & = [] \\ code(I_1\ A;) & = code(I_1) \mid code(A) \\ code(id{:=}E) & = code(E) \mid \textbf{STOREG}\ adr(id) \end{array}$$

Dans le cas d'une donnée à stocker sur k mots, on utilisera pour coder $id{:=}E$, les instructions :

$$code(E) \mid \textbf{STOREG}\ adr(id)+(k-1) \mid \ \ldots\ \mid \textbf{STOREG}\ adr(id)$$

L'invariant à conserver est que le code associé à une suite d'instructions démarre et se termine avec un pointeur de pile au dessus de l'ensemble des valeurs globales.

Tableaux

En général le nombre de cases de la pile dans lesquelles la donnée est représentée dépend du type de la donnée (entier, réel, tableau, record), on suppose cette taille connue à la compilation et notée *taille(id)*.

Les tableaux sont représentés par des suites de cases adjacentes dans la pile. Si le tableau t a des indices compris entre m et M, démarre à l'adresse a et contient des éléments de taille k alors il occupe une place de $(M-m+1)\times k$.

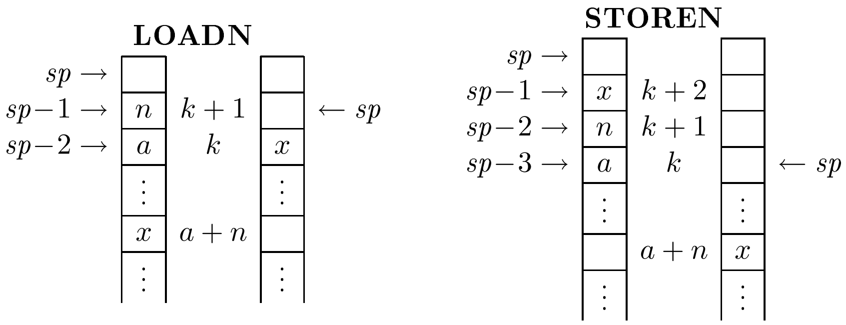
Pour calculer l'adresse correspondante à une expression $t[E]$, il faut calculer la valeur n de E puis accéder à l'adresse $a+(n-m)\times k$. Si on connaît la valeur de m à la compilation alors on peut partiellement évaluer cette expression en pré-calculant $a-m\times k$ et gardant cette valeur dans $t.val$. Il suffira ensuite d'effectuer l'opération $t.val+n\times k$.

Les commandes **PUSHG,STOREG** ne permettent que d'accéder à une adresse connue à la compilation, pour l'accès à des tableaux nous aurons besoin d'un accès fonction d'une valeur calculée à l'exécution.

En général, l'adresse à laquelle nous avons besoin d'accéder dépend d'une adresse de base a qui n'est pas forcément connue statiquement (dans le cas de tableaux de taille variable) et d'un décalage n qui est également calculé à l'exécution. On étend donc les commandes **PUSHG,STOREG** qui travaillaient à partir d'une adresse fixe *gp* et d'un décalage statique n pris en argument, en des commandes **LOADN** et **STOREN** qui prennent ces informations dynamiquement sur la pile.

Code	Pile	sp	pc	Condition
LOADN	$\mathsf{P}[sp-1]:=\mathsf{P}[\mathsf{P}[sp-2]+\mathsf{P}[sp-1]]$	$sp-1$	$pc+1$	
STOREN	$\mathsf{P}[\mathsf{P}[sp-3]+\mathsf{P}[sp-2]]:=\mathsf{P}[sp-1]$	$sp-3$	$pc+1$	

On peut représenter graphiquement le comportement de ces deux instructions en matérialisant la pile dont on numérote les cases par les entiers $k, k+1 \ldots$:



On remarque que ces instructions pourraient être décomposées en une instruction permettant d'accéder à un élément de la pile par l'intermédiaire d'une adresse stockée dans la pile, et une instruction permettant de faire des calculs sur les adresses.

On ajoute à notre langage des tableaux que l'on suppose unidimensionnels. On a donc deux nouvelles productions :

$$\begin{array}{lcl} E & ::= & id[E] \\ I & ::= & \mathsf{id}[E_1]:=E_2 \end{array}$$

Pour engendrer le code, il faut pouvoir mettre dans la pile une adresse. On se donne donc une instruction **PUSHGP** qui permet de stocker l'adresse du pointeur global dans la pile.

Code	Pile	sp	pc	Condition
PUSHGP	$\mathsf{P}[sp]:=gp$	$sp+1$	$pc+1$	

On suppose que l'on connaît l'adresse de base où est stocké chaque tableau. Le code engendré est :

$$\begin{array}{ll} code(id[E]) & = \textbf{PUSHGP} \mid \textbf{PUSHI}\ adr(id) \mid code(E) \mid \textbf{ADD} \mid \textbf{LOADN} \\ code(\mathsf{id}[E_1]:=E_2) & = \textbf{PUSHGP} \mid \textbf{PUSHI}\ adr(id) \mid code(E_1) \mid \textbf{ADD} \mid code(E_2) \mid \textbf{STOREN} \end{array}$$

Les tableaux bi-dimensionnels peuvent être stockés soit en lignes soit en colonnes. Si chaque dimension a pour indice minimal m_i et maximal M_i et pour taille $n_i = M_i - m_i + 1$ si le stockage se fait en ligne alors on aura d'abord le tableau $t[1, i]$ puis $t[2, i]$ …L'élément $t[i_1, i_2]$ est stocké à l'adresse

$$a + ((i_1 - m_1) \times n_2 + (i_2 - m_2)) \times k$$

Là-aussi on peut pré-calculer une partie de l'expression en réécrivant cela sous la forme :

$$((i_1 \times n_2 + i_2) \times k + a - (m_1 \times n_2 + m_2) \times k$$

Lorsqu'on rencontre un expression $id[E_1, E_2]$ il faudra calculer le décalage à appliquer à partir de la base où est stockée id pour accéder à l'élément désigné.

On peut généraliser cela aux tableaux de dimension p. Dans le cas de tableau dont la taille n'est pas connue à la compilation, on ne pourra pas effectuer de pré-compilation.

Autres instructions d'accès

On peut également avoir besoin d'accéder à une donnée sur la pile référencée de manière statique à partir d'une adresse stockée sur la pile. C'est le rôle des instructions **LOAD** et **STORE**

Code	Pile	sp	pc	Condition
LOAD n	$\mathsf{P}[sp-1] := \mathsf{P}[\mathsf{P}[sp-1]+n]$	sp	$pc+1$	n entier
STORE n	$\mathsf{P}[\mathsf{P}[sp-2]+n] := \mathsf{P}[sp-1]$	$sp-2$	$pc+1$	n entier

6.3.5 Instructions conditionnelles

Pour compiler des expressions conditionnelles et des boucles on aura besoin de commandes pour effectuer des sauts dans la zone d'instructions. Les instructions seront désignées par des adresses symboliques insérées dans le code.

Code	Pile	sp	pc	Condition
JUMP *label*		sp	*label*	
JZ *label*		$sp-1$	si $\mathsf{P}[sp-1]=0$ alors *label* sinon $pc+1$	

Pour compiler les expressions, il faut adopter une convention pour la représentation des booléens. Il y a différentes options, 0 peut représenter *faux* et pour *vrai*, on peut prendre les entiers positifs ou utiliser seulement 1.

On regarde comment engendrer le code pour des expressions conditionnelles et des boucles :

$$\begin{array}{lcl} I & ::= & \mathsf{if}\ E\ \mathsf{then}\ I\ \mathsf{endif} \\ I & ::= & \mathsf{if}\ E\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2\ \mathsf{endif} \\ I & ::= & \mathsf{while}\ E\ \mathsf{do}\ I\ \mathsf{done} \end{array}$$

On introduit une commande supplémentaire du langage intermédiaire **LABEL** *label* qui introduit un adresse symbolique *label* correspondant au numéro de l'instruction sans modifier l'état de la pile. Seul le compteur d'instructions est incrémenté.

Exercice Donner un système d'attributs pour le langage intermédiaire permettant de supprimer les instructions **LABEL** et de remplacer les adresses symboliques par des adresses entières.

$$\begin{array}{ll} code(\mathsf{if}\ E\ \mathsf{then}\ I\ \mathsf{endif}) & = code(E) \mid \mathbf{JZ}\ \textit{new-suiv} \mid code(I) \mid \mathbf{LABEL}\ \textit{new-suiv} \\ code(\mathsf{if}\ E\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2\ \mathsf{endif}) & = code(E) \mid \mathbf{JZ}\ \textit{new-faux} \mid code(I_1) \mid \mathbf{JUMP}\ \textit{new-suiv} \\ & \ \mathbf{LABEL}\ \textit{new-E.faux} \mid code(I_2) \mid \mathbf{LABEL}\ \textit{new-suiv} \\ code(\mathsf{while}\ E\ \mathsf{do}\ I\ \mathsf{done}) & = \mathbf{LABEL}\ \textit{new-loop} \mid code(E) \mid \mathbf{JZ}\ \textit{new-suiv} \mid code(I) \mid \\ & \ \mathbf{JUMP}\ \textit{new-loop} \mid \mathbf{LABEL}\ \textit{new-suiv} \end{array}$$

Pour cette compilation il faut créer à chaque fois des nouvelles étiquettes *new-faux,new-loop, new-suiv.*

Compilation d'expressions booléennes Les expressions booléennes servant souvent à des opérations de contrôle, on peut se donner a priori deux étiquettes *E-vrai* et *E-faux* et compiler l'expression booléenne sans calculer de valeur mais en décidant que le résultat de l'exécution de E doit aboutir à $pc = \textit{E-vrai}$ lorsque la valeur de E est *vrai* et à *E-faux* sinon. On définit donc une fonction *code-bool* qui prend comme argument l'expression booléenne et les deux étiquettes et qui renvoie le code de controle.

Soit une grammaire d'expressions booléennes :

$$
\begin{array}{lll}
B & := & B_1 \textsf{ or } B_2 \\
B & := & B_1 \textsf{ and } B_2 \\
B & := & \textsf{not } B_1 \\
B & := & \textsf{vrai} \\
B & := & \textsf{faux} \\
B & := & E_1 \textsf{ relop } E_2
\end{array}
$$

Le code engendré correspondant est :

$$
\begin{array}{ll}
\textit{code-bool}(B_1 \textsf{ or } B_2, e_v, e_f) & = \textit{code-bool}(B_1, e_v, \textit{new-e}) \mid \mathbf{LABEL}\ \textit{new-e} \mid \textit{code-bool}(B_2, e_v, e_f) \\
\textit{code-bool}(B_1 \textsf{ and } B_2, e_v, e_f) & = \textit{code-bool}(B_1, \textit{new-e}, e_f) \mid \mathbf{LABEL}\ \textit{new-e} \mid \textit{code-bool}(B_2, e_v, e_f) \\
\textit{code-bool}(\textsf{not } B_1, e_v, e_f) & = \textit{code-bool}(B_1, e_f, e_v) \\
\textit{code-bool}(\textsf{vrai}, e_v, e_f) & = \mathbf{JUMP}\ e_v \\
\textit{code-bool}(\textsf{faux}, e_v, e_f) & = \mathbf{JUMP}\ e_f \\
\textit{code-bool}(E_1 \textsf{ relop } E_2, e_v, e_f) & = \textit{code}(E_1) \mid \textit{code}(E_2) \mid \textit{code}(\textsf{relop}) \mid \mathbf{JZ}\ e_f \mid \mathbf{JUMP}\ e_v
\end{array}
$$

Dans ce code, *new-e* représente une étiquette nouvellement créée. *code*(relop) représente l'instruction de comparaison correspondant à l'opérateur relationnel utilisé.

Remarque Attention dans un langage avec effets de bord dans les expressions le fait de calculer ou non les sous-expressions d'une expression booléenne peut avoir des comportements très différents.

Compilation des expressions avec branchements multiples

Il s'agit d'expressions switch ou case permettant des branchements multiples suivant les différentes valeurs d'une expression, ces valeurs, en général dans un type numérique, étant connues à la compilation :

$$
\begin{array}{lll}
\textsf{switch } E & \textsf{begin} & \textsf{case } V_1 : S_1 \\
 & & \textsf{case } V_2 : S_2 \\
 & & \ldots \\
 & & \textsf{case } V_n : S_n \\
 & \textsf{end} &
\end{array}
$$

Il faut d'abord s'assurer de la sémantique d'une telle expression. Une sémantique naturelle est :

$$
\begin{array}{lll}
\textsf{if } E = V_1 & \textsf{then} & S_1 \\
\textsf{else if } E = V_2 & \textsf{then} & S_2 \\
\ldots & & \\
\textsf{else if } E = V_n & \textsf{then} & S_n
\end{array}
$$

Cependant ce n'est pas la sémantique de l'instruction `switch` dans C, dont la sémantique est que l'on exécute toutes les instructions $S_i; \ldots; S_n$ à partir du premier i tel que $E = V_i$, c'est la présence d'une instruction `break` à la fin de S_i qui permet de sortir du `switch`.

La sémantique est :

$$
\begin{array}{lll}
\textsf{if } E = V_1 & \textsf{then} & \textsf{goto } l_1 \\
\textsf{else if } E = V_2 & \textsf{then} & \textsf{goto } l_2 \\
\ldots & & \\
\textsf{else if } E = V_n & \textsf{then} & \textsf{goto } l_n \\
l_1 : & & S_1 \\
\ldots & & \\
l_n : & & S_n \\
\text{break :} & &
\end{array}
$$

On peut optimiser la compilation si les valeurs des V_i sont toutes distinctes dans un intervalle $[k+1, k+n]$. On crée alors dans le code une table de branchements qui débute par une étiquette **LABEL** *debut* et qui comporte successivement les instructions : **JUMP** $l_1 \ldots$ **JUMP** l_n. Suivent ensuite les codes de chaque branche : LABEL l_i | $code(S_i)$ suivi éventuellement d'une instruction **JUMP** *fin*. Il est alors nécessaire d'avoir une instruction de saut indexé que nous noterons **JUMPI**. Cette instruction prend un label et renvoie le contrôle à l'instruction dont le numéro est la valeur numérique du label plus la valeur du sommet de la pile.

Code	Pile	sp	pc	Condition
JUMPI *label*		$sp-1$	$label + \mathsf{P}[sp-1]$	

On insère alors le code de E : $code(\mathrm{E})$ auquel il faut soustraire la constante k : **PUSHI** k | **SUB**. On teste si la valeur obtenue est bien comprise entre 1 et n et on effectue le branchement indexé correspondant, sinon l'instruction est sautée. Comme la valeur en sommet de pile doit être utilisée pour deux comparaisons et le saut indexé, il faut l'utiliser 3 fois et donc la dupliquer deux fois à l'aide de l'instruction **DUP**. On obtient le code suivant :

DUP | **DUP**
| **PUSHI** n | $INFEQ$ | **JZ** fin
| **PUSHI** 1 | $SUPEQ$ | **JZ** fin
| **JUMPI** debut
| **LABEL** fin

Exercice Modifier ce schéma dans le cas où l'instruction switch comporte un cas par défaut.

6.3.6 Appels de procédure

L'appel de procédure a deux effets, le premier est de modifier le cours d'exécution des instructions le second est de créer un espace pour des données locales.

Sous-routines

Supposons que l'on veuille réutiliser à plusieurs endroits la même séquence de code I, on peut vouloir la partager. Pour cela on isole cette partie du code et on lui donne une étiquette. On veut remplacer la copie de la suite d'instructions I par un saut du compteur de programme vers le code de p. Le problème est d'indiquer où revenir à la fin de l'exécution de la séquence. Si on a plusieurs utilisations de la routine alors on ne peut pas mettre à la fin du code de I une simple étiquette de saut statique.

Il faut donc avant l'appel de la routine sauvegarder le compteur de programme et le restaurer à la sortie de l'appel. Comme plusieurs appels de sous-routines peuvent s'emboîter, il faut sauvegarder un nombre arbitraire de telles valeurs.

Cette information pourrait être stockée dans la pile des données, dans la machine que nous utilisons il y a une pile indépendante pour stocker les compteurs.

Les instructions **CALL,RETURN** permettent de gérer la modification du pointeur de programme. L'instruction **CALL** prend comme argument une adresse dans le code d'instructions. Le compteur d'instructions se place alors à cette adresse, son ancienne valeur est sauvegardée. L'instruction **RETURN** retrouve l'ancienne valeur du compteur d'instructions et se place à l'instruction suivante.

Nous ajoutons des définitions et appels de routines dans notre langage :

$$\begin{array}{lcl} D & ::= & \mathsf{proc}\ p;\ \mathsf{begin}\ I\ \mathsf{end} \\ I & ::= & \mathsf{call}\ p \end{array}$$

$$\begin{array}{ll} code(\mathsf{proc}\ p;\ \mathsf{begin}\ I\ \mathsf{end}) & = \textbf{LABEL}\ \textit{label-p} \mid code(I) \mid \textbf{RETURN} \\ code(\mathsf{call}\ p) & = \textbf{CALL}\ \textit{label-p} \end{array}$$

label-p est une étiquette unique associée à la procédure p.

Procédures avec paramètres

Si la procédure a des paramètres alors le rôle de cette procédure est de permettre d'allouer dans la pile la place pour ces variables et de les référencer. Comme on ne sait référencer que par rapport à la base, il faut ajouter un registre *fp* qui sera mis à jour au début de l'appel de procédure et qui permettra de référencer les valeurs locales. A la sortie de la procédure cet espace doit être libéré.

L'appel et le retour des procédures revient donc à une modification de la valeur de *pc* et l'affectation d'une adresse de base pour les variables locales. Au retour de la procédure la valeur de *pc* doit augmenter de 1 et celle de *fp* être restituée à la valeur qu'elle avait avant l'appel puisque les appels peuvent être imbriqués. La pile sera tronquée de toute la partie allouée après l'appel à la procédure.

Le rôle des commandes **CALL** et **RETURN** va justement être d'assurer la sauvegarde et la restauration des registres *fp* et *pc*.

Instructions de manipulation de données locales Notre langage contient des instructions pour manipuler des données référencées à partir du pointeur *fp*.

Aux instructions **PUSHGP, STOREG, PUSHG** correspondent les instructions **PUSHFP, STOREL, PUSHL** qui ont le même comportement en remplaçant *gp* par *fp*.

Tableau d'activation Dans le cas d'une allocation statique, le tableau d'activation va être constitué des paramètres de la procédure qui sont instanciés à l'appel, de l'adresse du tableau d'activation de la procédure emboîtante et des emplacements pour les variables locales.

La grammaire pour les déclarations a la forme suivante :

$$\begin{array}{lll} Ds & ::= & Ds\ D;\ \mid \epsilon \\ D & ::= & \mathsf{var}\ id : T \mid \mathsf{proc}\ id(Ds)\ Ds\ \mathsf{begin}\ I\ \mathsf{end} \mid \mathsf{fun}\ id(Ds) : T\ Ds\ E \end{array}$$

On peut calculer (par exemple) par des attributs, le niveau de chaque symbole de procédure, fonction ou variable qui est le nombre de procédures ou fonctions sous lequel il est défini et qui correspond à la profondeur du symbole dans l'arbre de déclarations. Le programme principal a un niveau égal à 0.

Le décalage associé à une variable est l'entier qu'il faut ajouter à *fp* pour trouver l'adresse de base de la variable.

Exercice Donner des systèmes d'attributs pour calculer les niveaux et décalages des déclarations

Au moment d'un appel par valeur de $q(e_1, \ldots, e_n)$ on va effectuer les codes de $e_1, \ldots, e_n$ qui vont être empilés, on va calculer l'adresse du tableau d'activation englobant. Pour cela on regarde les niveaux respectifs de q et de la procédure p qui a appelé q. La donnée importante est le niveau relatif n qui est égal au niveau de p moins le niveau de q et qui est toujours supérieur ou égale à -1 (=-1 lorsque q est déclaré dans p).

Calcul de l'adresse du tableau d'activation père Lorsque p appelle q alors le calcul de l'adresse du père de q se fait par la suite d'instructions :

$$\textbf{PUSHFP} \mid \underbrace{\textbf{LOAD}\ -1}_{n+1\ \text{fois}}$$

On peut remarquer que la suite d'instructions **PUSHFP** | **LOAD** n est équivalente à **PUSHL** n.

Lorsqu'on a mis à jour ce registre on peut lancer l'appel à la commande **CALL** q. Le début de cette procédure doit réserver la place pour les variables locales puis exécuter le code. À la fin de l'exécution de la procédure la commande **RETURN** est appelée. Celle-ci supprimera de la pile toutes les variables locales mais pas les paramètres.

L'appelant peut alors effectuer une opération de dépilement des paramètres ainsi que de l'adresse du père.

Si la procédure est une fonction qui renvoie une valeur, alors l'emplacement de cette valeur devra être réservé avant l'appel de procédure. On choisit en général d'utiliser le premier emplacement dans la pile.

Calcul de l'accès à une variable Si dans une procédure p on veut accéder à une variable x on calcule encore une différence n entre le niveau np de p et celui nx de x qui est supérieure à -1. On connaît par ailleurs le décalage dx de x.

Le code pour accéder à la valeur droite de x est donc paramétré par le niveau np de la procédure dans laquelle on est :

$$\textit{code-var}(x, np) = \mathbf{PUSHFP} \mid \underbrace{\mathbf{LOAD}\ -1}_{np-nx+1 \text{ fois}} \mid \mathbf{LOAD}\ dx$$

Passage de paramètres par valeur Dans un mode de passage de paramètres par valeur on aura la compilation suivante du corps d'une procédure :

$$\textit{code}(\mathsf{proc}\ id(Ds_1)Ds_2\ \mathsf{begin}\ I\ \mathsf{end}) = \mathbf{LABEL}\ \textit{label-id} \mid \mathbf{PUSHN}\ \textit{taille}(Ds_2) \mid \textit{code}(I) \mid \mathbf{RETURN}$$

L'appel se fait par les instructions :

$$\begin{array}{lll} LE & ::= & E \mid LE, E \\ A & ::= & \mathsf{call}\ id(LE) \end{array}$$

La génération de code d'une suite d'expressions est juste la concaténation des codes. Le code associé à l'appel d'une procédure q de niveau nq est paramétré par le niveau np de la procédure appelante :

$$\begin{array}{rl} \textit{code-call}(\mathsf{call}\ q(LE), np) = & \textit{code}(LE) \mid \mathbf{PUSHFP} \mid \underbrace{\mathbf{LOAD}\ -1}_{np-nq+1 \text{ fois}} \\ & \mid \mathbf{CALL}\ \textit{label-id} \mid \mathbf{POP}\ \textit{taille}(LE) + 1 \end{array}$$

Exercice Écrire le code de compilation et d'appel d'une fonction.

Passage de paramètres par référence Au lieu d'empiler les valeurs des expressions, on empile leur valeur gauche (c'est-à-dire une adresse). Les accès aux variables se font donc par une indirection.

Passage d'une fonction en paramètre Pour compiler une procédure qui prend une fonction en paramètre, il faut deux informations, l'adresse du code de la fonction et l'adresse du tableau d'activation du père de la procédure.

Exemple

```
program main
procedure b(function h(n:int):inte);
begin print(h(2)) end

procedure c;
var m:int;
function f(n:int) : int; begin f:=m+n end
begin m:=0; b(f) end

begin c end
```

Pour exécuter le code de la procédure, il est nécessaire de connaître l'adresse du code de cette procédure ainsi que celle du tableau d'activation de la procédure

Lorsque c doit exécuter b(f) il calcule le lien d'activation du père statique de f (ici c) comme s'il l'appelait et le passe à b qui pourra l'utiliser au moment de l'appel de f.

Renvoyer une fonction comme valeur Certains langages permettent de renvoyer des fonctions comme valeurs. Cependant si le langage autorise l'accès à des variables non locales, le problème se pose de la persistance des valeurs ainsi manipulées.

Exemple

```
let f(x) = let g(y) = x+y in g
let h = f(3)
let j = f(4)
h(5)+j(7)
```

La valeur d'une fonction est appelée une *clôture* elle est formée d'un pointeur à l'adresse du code de la fonction ainsi que d'un environnement qui donne les valeurs de toutes les variables utilisées dans le corps de la fonction et qui ne sont pas locales au moment de la définition de la fonction.

6.4 Utilisation des registres

Les registres permettent de stocker des informations et d'y accéder de manière rapide. Cependant les registres sont en nombre limités. Il convient donc de les exploiter au mieux.

Une manière de réaliser cela est d'introduire pour chaque valeur intermédiaire une nouvelle variable et de construire un graphe d'interférence : les nœuds sont les variables et on a une arête entre x et y si x et y ne peuvent être sauvées dans le même registre (ie x et y sont simultanément vivantes). On peut également introduire les registres comme nœuds de ce graphe et exprimer par une arête le fait que certaines valeurs ne peuvent pas être stockées dans un registre particulier.

Déterminer une répartition des variables sur k-registres revient à trouver un k-coloriage du graphe. Comme ce problème est NP-complet en général, on utilise une approximation.

Si le graphe est vide, alors le coloriage est évident. Si le graphe comporte un nœud x de degré strictement inférieur à k, alors on construit un nouveau graphe G' en retirant ce nœud et les arêtes correspondantes, si G' peut être colorié, alors on peut trouver une couleur pour x, différente de celle de ses voisins et l'ajouter. On remarque que retirer x de G peut faire diminuer le degré d'autres nœuds et rendre ainsi le coloriage possible. Si le graphe ne comporte que des nœuds de degré supérieur à k alors on choisit un nœud x que l'on retire et on cherche à colorier le graphe résultant. Si c'est possible, on regarde le nombre de couleurs utilisées par les voisins de x. S'il en reste une de libre alors il est possible de colorier x et on l'ajoute au graphe. Sinon, le coloriage a échoué, on décide donc de stocker x dans la mémoire. Pour cela, on choisit un emplacement mémoire m_x pour stocker x et on introduit pour chaque utilisation de x dans le programme une nouvelle variable x_i. Si la valeur de x est utilisée dans une expression, on commencera par mettre dans x_i la valeur stockée à l'adresse m_x de la mémoire. Si x est mise à jour alors on remplace x par x_i dans la mise à jour, et on fait suivre cette instruction d'une mise à jour de m_x dans la mémoire par la valeur x_i. La durée de vie des variables x_i est courte, elles n'interfèrent pas avec les autres variables.

Il faut alors modifier le graphe d'interférence et recommencer le coloriage.

Le graphe d'interférence peut également être utilisé pour supprimer des instructions **move** entre registres associées à des instructions `x :=y`. On peut identifier x et y dès lors que x et y ne sont pas vivantes simultanément. Cependant, cette identification aura pour résultat d'augmenter le degré du nœud xy résultat de l'identification de x et de y ce qui pourrait faire échouer le coloriage. Or effectuer un déplacement dans les registres est moins coûteux que d'accéder à la mémoire. On ne fera donc l'identification que lorsque l'on garantit la préservation du coloriage. C'est le cas si les voisins de xy comportent strictement moins de k nœuds de degré supérieur à k.

Exemple (d'après [3]). On considère le programme suivant formé d'affectations simples et d'accès à la mémoire. On indique dans le seconde colonne l'ensemble des variables simultanément vivantes au point de programme précédent l'instruction, sachant qu'après l'exécution de ce code les variables d, k et j sont vivantes.

instruction	variables vivantes
g := mem [j+12]	k, j
h := k - 1	j, g, k
f := g * h	j, g, h
e := mem[j+8]	j, f
m := mem[j+16]	j, f, e
b := mem[f]	m, f, e
c := e+8	m, b, e
d := c	m, b, c
k := m+4	m, b, d
j := b	b, d, k
	d, k, j

On peut examiner les nœuds du graphe dans l'ordre : $m, c, b, f, e, j, d, k, h, g$ on constate qu'à chaque instant, chaque nœud a un degré inférieur ou égal à 3. On peut donc faire un coloriage avec 4 couleurs et on obtient par exemple l'affectation suivante : $m = 1, c = 2, b = 3, f = 2, e = 4, j = 3, d = 4, k = 1, h = 1, g = 2$.

Si on ne disposait que de 3 registres, on pourrait refaire l'analyse en examinant les nœuds dans l'ordre suivant. Les nœuds indiqués en gras sont ceux dont le degré est supérieur à 3 et donc qui peuvent poser un problème. $b, d, j, e, f, \mathbf{m}, \mathbf{k}, g, c, h$. Une tentative de coloriage $b = 1, d = 2, j = 1, e = 2, f = 3, \mathbf{m}, \mathbf{k}, g, c, h$ ne permet pas de colorier m. On peut donc choisir de stocker m en mémoire à l'adresse `M`. On doit donc transformer le programme :

instruction		variables vivantes
g	:= mem [j+12]	k, j
h	:= k - 1	j, g, k
f	:= g * h	j, g, h
e	:= mem[j+8]	j, f
m_1	:= mem[j+16]	j, f, e
mem[`M`]	:= m_1	m_1,f,e
b	:= mem[f]	f, e
c	:= e+8	b, e
d	:= c	b, c
m_2	:= mem[`M`]	b, d
k	:= m_2+4	m_2, b, d
j	:= b	b, d, k
		d, k, j

Le nœud m de degré 5 s'est transformé en deux nœuds m_1 et m_2 de degré 2. On peut réexaminer les nœuds du nouveau graphe dans l'ordre suivant : $h, c, m_2, m_1, f, e, b, d, k, g, j$ et il n'y a plus de nœud à problème.

Un principe de coloriage peut aussi être utilisé pour minimiser le nombre de places mémoires des variables stockées en mémoire. Un objectif sera alors de minimiser le nombre de `move` même si cela doit augmenter le nombre de cases mémoire utilisées.

6.5 Allocation dans le tas

Dans le codage que nous venons de voir, les nouvelles valeurs à calculer au cours de l'exécution sont allouées dans la pile. Ce principe fonctionne car ce qui est alloué au cours de l'appel d'une procédure n'est plus accessible à la fin de la procédure.

Ce principe ne permet pas de couvrir toutes les constructions de programme. Par exemple, si le langage permet de renvoyer des valeurs fonctionnelles, alors on a vu que la valeur de cette fonction doit contenir une représentation de l'environnement dans lequel la fonction est définie. Cet environnement qui survit à la fin de la procédure qui définit la fonction comprend pourtant des valeurs allouées dans le tableau d'activation de la procédure qui a défini la fonction.

6.5.1 Représentation des données structurées

Dans les exemples précédents, nous avons alloué pour chaque variable, un espace correspondant à la taille de cette donnée. Ainsi le passage de tableaux en paramètres nécessite de les recopier. Il serait plus efficace de manipuler les objets de taille volumineuse par l'intermédiaire de leur adresse. Dans le cas de tableaux dans CAML par exemple, il y a un moment où le tableau est créé (explicitement par `[|1;2;3|]` ou via la commande `Array.create`) puis une valeur de type tableau est simplement l'adresse d'un tableau. Évidemment, le tableau peut être alloué dans le corps d'une fonction et renvoyé par exemple comme résultat de la fonction, il faut donc qu'il soit alloué dans une zone de données persistante.

Allouer de telles données dans la zone des variables globales n'est pas forcément la meilleure stratégie. En effet, la durée de vie de tels objets est en général limitée, il est donc inutile de réserver de la place mémoire tout au long de l'exécution du programme.

6.5.2 Allocation dynamique

Il s'agit de créer dynamiquement au cours du programme de nouveaux espaces mémoires qui seront accessibles par le programme sans forcément être directement liés à une variable. La durée de vie de ces objets n'est pas forcément limitée à la durée de vie des procédures. On réserve donc l'espace pour ces objets en général dans le tas. Il est possible que des cases mémoires ainsi réservées ne puissent plus être accessibles en cours de programme.

On distingue dans le langage, les données directes telles que les entiers qui seront allouées et manipulées dans la pile des données dont la manipulation se fera par l'intermédiaire d'une adresse dans la mémoire. Une telle manipulation via des adresses est nécessaire lorsque le langage contient des fonctions *polymorphes* qui peuvent manipuler des données de taille variable à l'exécution.

6.5.3 Allocation/desallocation explicite

Les langages comme Pascal ou C offrent des primitives (new/dispose en Pascal) pour allouer ou désallouer des parties de la mémoire.

Si toutes les données à allouer sont de même taille alors il est possible d'en faire une liste chaînée dans laquelle on va chercher un nouvel espace libre ou bien on va rendre une case. On réserve un espace qui comprend l'espace nécessaire à la donnée à allouer plus l'adresse du suivant sur la pile des espaces disponibles.

Si on doit allouer des objets de taille différente alors il faudra utiliser le tas. La désallocation d'une donnée ne libère pas forcément un espace en mémoire suffisant pour l'allocation des données suivantes, le tas se transforme alors en gruyère, même si l'espace libre nécessaire est suffisant, il peut être trop fractionné pour permettre l'allocation.

L'allocation/ désallocation explicite rend la programmation plus lourde et fait courir le risque soit d'utiliser inutilement la place par des données inaccessibles soit au contraire de faire disparaître des données accessibles au risque de provoquer des erreurs difficilement détectables ensuite suivant la manière dont cet espace est ré-alloué.

6.5.4 Allocation/desallocation implicite

Des langages tels que LISP, ML ou JAVA ont choisi la voie de l'allocation dynamique. L'exécution va être amenée à allouer de la mémoire et s'occupera également de sa récupération, c'est le mécanisme de Garbage collector (ramasse-miette, glaneur de cellule).

Il existe plusieurs méthodes pour récupérer la mémoire.

Compteur de référence

Chaque cellule dynamiquement allouée comporte un compteur indiquant combien de pointeurs peuvent y accéder.

Lorsque l'on fait new(p) alors la cellule a un compteur initialisé à 1. Une affectation entre pointeurs à la forme $t := u$: t est une expression qui doit s'évaluer en une valeur gauche x qui représente un pointeur et dont la valeur droite est une adresse vers une cellule m. L'expression u qui représente également un pointeur s'évalue en une valeur droite qui est une adresse vers une cellule n. La cellule m qui était précédemment accessible par x ne l'est plus son compteur décroit de 1. Par contre la cellule n est maintenant accessible à partir de x donc son compteur est incrémenté de 1. Les cellules dont le compteur est zéro peuvent être récupérées, par contre il est possible que deux cellules pointent mutuellement chacune sur l'autre et donc gardent des compteurs à 1 même si elles ne sont plus accessibles par le programme. Cette méthode est de plus coûteuse en temps de calcul.

Exemple

```
type list = Nil | Cons of int * list ref
let x = Cons(7,ref Nil)
```

```
in let y = Cons(9,ref x)
in let t = Cons(10,ref y)
in let Cons(_,z)=x in z:=y;;
```

Marquer et Balayer

Il s'agit de partir des variables de la pile qui sont accessibles au programme et de marquer toutes les cellules ainsi utilisables de manière récursive. Lorsque ce procédé est terminé, une deuxième passe permet de libérer tous les espaces inaccessibles.

Cette récupération de mémoire est coûteuse en temps ce qui pose un problème pour les applications temps réel. Elle a également l'inconvénient de fragmenter la mémoire. D'autre part elle utilise un procédé récursif pour marquer la mémoire ce qui pourrait provoquer un dépassement de capacité mémoire.

Pour éviter d'utiliser de la place mémoire supplémentaire pour gérer les appels récursifs, on peut effectuer un parcours en profondeur des liens de la mémoire et inverser les liens entre les cases mémoire pour se rappeler des zones qu'il reste à parcourir.

Stopper et Copier

Pour éviter les problèmes de fragmentation on peut décider de toujours allouer linéairement des données. Lorsque la zone est pleine on s'arrête et on recopie toute la partie utile dans une seconde zone de manière continue, on libère toute la zone précédemment occupée.

Cette méthode a l'inconvénient de ne pouvoir effectivement occuper que la moitié de l'espace réservé.

Méthodes mixtes

Les GC modernes utilisent un compromis entre ces différentes méthodes. Par exemple caml-light utilise un GC mis au point par Damien Doligez. L'espace mémoire est séparé en deux zones l'une dite vieille organisée à l'aide d'une liste de places accessibles cette zone pourra elle-même être étendue dynamiquement si nécessaire, l'autre dite jeune organisée comme un tableau linéaire de taille fixe. Les cellules sont allouées linéairement dans le tableau. Lorsque ce tableau est plein on stoppe et on copie les objets utiles (ceux accessibles à partir des variables de la pile, des registres ou des objets de la vieille zone, ces derniers étant conservés dans une table de références) dans la vieille zone. Cette étape s'appelle un gc mineur.

On va ensuite faire un gc majeur qui va consister en un marquage/balayage incrémental de la vieille mémoire. Pour marquer les objets on utilise un coloriage. Les nœuds sont blancs quand ils n'ont pas été visités, gris quand ils ont été visités mais pas leurs fils et noir quand ils ont été visités ainsi que leurs fils immédiats. On garde une pile des nœuds gris. Lorsqu'il ne reste plus de nœuds gris alors on a fini le marquage. Les nœuds qui sont blancs après le marquage peuvent être rendus pour la liste libre de la vieille génération.

Afin de gérer le GC, la représentation de CAML réserve un bit sur chaque mot pour distinguer les adresses (que le GC doit suivre) des entiers. Le tas est organisé en zones correspondant aux différentes structures allouées (tableau, type de données structuré, clôture,...). Dans l'entête de chaque zone on réserve deux bits pour le marquage (blanc,gris,noir) du GC.

Chapitre 7

Compilation d'un langage objet

7.1 Introduction

Nous allons étudier les caractéristiques de la compilation d'un langage objet. Ces langages sont très populaires car ils offrent des mécanismes de structuration et d'encapsulation des données appropriés au développement de larges applications. Cependant nous verrons que les mécanismes sophistiqués des langages objets posent des problèmes d'efficacité dans la compilation.

7.2 Compilation des objets

La déclaration d'une classe comporte des variables d'états et des méthodes, celles-ci peuvent éventuellement être déclarées statiques, auquel cas elles se comporteront comme des variables globales et des procédures ordinaires.

Un objet correspond à une zone mémoire comportant les différentes variables d'état. Cette zone est allouée au moment de la création de l'objet. Les méthodes se comportent comme des procédures : elles ont des arguments déclarés et éventuellement un résultat. Par défaut, elles s'appliquent à l'objet lui même (qui est référencé par exemple par le mot clé `this`).

7.2.1 Visibilité

Les règles de visibilité des langages objet restreignent l'accès aux données. Ainsi les variables d'état privées introduites dans une classe ne seront visibles que dans les définitions de sous-classes. De même, des méthodes déclarées privées ne peuvent être utilisées en dehors de la classe.

7.2.2 Typage

Ce sont les classes des langages objets qui jouent le rôle des types. Au centre des langages objets, se trouve la notion d'héritage. Une classe C peut hériter d'une classe D. Les variables d'état de D seront alors également des variables d'état de C et les méthodes de D pourront s'appliquer aux objets de la classe C. Une méthode de D peut également être redéfinie dans C, il y a alors surcharge du nom de la méthode : le même nom étant associé à plusieurs codes. Cette ambiguïté devra être résolue soit à la compilation soit à l'exécution.

Typage statique ou dynamique

Il y a deux manières de désambiguer le choix d'une méthode à appliquer :

- On détermine au *typage* la classe de l'objet et ceci permet de connaître *statiquement* la méthode à appliquer.

– C'est la classe de l'objet au moment de *l'exécution* qui détermine dynamiquement la méthode à sélectionner.

Les classes des objets jouent un rôle important dans la compilation, aussi les objets manipulés comportent explicitement un pointeur vers le descripteur de leur classe. Le descripteur de classe comporte les informations utiles sur la classe, comme les labels associés aux méthodes définies dans la classe, les classes ancêtres, la taille des variables d'état.

Héritage simple

Dans le cas de l'héritage simple, une classe C hérite au plus d'une autre classe D. Ceci permet d'organiser simplement les variables d'états des objets de la classe C, comme une extension des variables de la classe D de manière à ce que les méthodes compilées pour les objets de la classe D puisse s'appliquer aux objets de la classe C. Un objet de la classe C qui hérite de D contient d'abord les variables de D puis celles de C.

Exemple

```
class A extends Object {int a=0;}
class B extends A {int b=0; int c=0;}
class C extends A {int d=0;}
class D extends B {int e=0;}
```

Les méthodes statiques peuvent être compilées comme des appels à des procédures car le label de la méthode à appliquer est connu au moment de la compilation.

Dans le cas de méthodes dynamiques, le descripteur de classe doit contenir l'adresse du code à appeler.

Exemple

```
class A extends Object {int x=0; void f(){...}}
class B extends A {void g(){...}}
class C extends B {void g(){...}}
class D extends C {int y=0; void f(){...}}
```

Pour compiler `c.f()` il faut

– Trouve le descripteur de classe de c
– Chercher le label associé au code de f
– Appeler le code associé

Le programme engendré comportera quatre fonctions correspondant aux code déclaré dans les classes, avec pour label `A_f`, `B_g`, `C_g` et `D_f`. Les descripteurs de classe sont stockés dans la pile à une adresse globale. Ils contiennent l'information sur les classes utiles à l'exécution : l'adresse du descripteur de la classe père (afin de pouvoir décider si un objet est une instance d'une autre classe) ainsi que les labels de chaque méthode applicable à un objet de cette classe. On a dans notre exemple :

	A
père	`Object`
f	`A_f`

	B
père	`A`
f	`A_f`
g	`B_g`

	C
père	`B`
f	`A_f`
g	`C_g`

	D
père	`C`
f	`D_f`
g	`C_g`

Chaque objet va être représenté en mémoire à l'aide d'une ensemble de cellules, une des cellules représente l'adresse du descripteur de la classe , les autres cellules contiennent les valeurs des attributs dynamiques.

	objet A		objet B		objet C		objet D
classe	`A`	classe	`B`	classe	`C`	classe	`D`
x		x		x		x	
						y	

La table des symboles fournit une information statique, utilisée par le compilateur mais qui ne sert pas à l'exécution. Elle peut contenir pour chaque classe déclarée les informations suivantes : adresse statique où sera stocké le descripteur de la classe, classe père, nom des attributs et décalage associé, nom des méthodes applicables et décalage associé dans le descripteur de classe.

La valeur d'un objet est l'adresse (dans le tas) de sa représentation.

On ne connaît plus statiquement l'adresse du code à appeler, cette adresse se trouve sur la pile.

Héritage multiple

Dans le cas de l'héritage multiple[1], une classe C peut hériter des objets des classes D_1 et D_2. On ne peut plus organiser simplement les variables de C, D_1 et D_2 pour que les procédures compilées pour D_i s'appliquent aux objets de C.

Exemple

```
class A extends Object {int a=0;}
class B extends Object {int b=0; int c=0;}
class C extends A {int d=0;}
class D extends A B C {int e=0;}
```

Il faut trouver pour chaque variable d'état un décalage qui soit le même pour toutes les sous-classes. Pour cela on construit un graphe dont les sommets sont les variables d'état et où on a une arête entre deux variables si elles coexistent au sein d'une même classe. Le problème est de colorier ce graphe avec le moins de couleurs possibles, chaque couleur correspondant à un décalage.

Si les objets étaient représentés en tenant compte de ces décalages alors il y aurait des trous dans la représentation. Aussi seul le descripteur de classe a la forme adéquate, chaque place renvoyant à la position où la variable est stockée dans l'objet. Cette représentation induit un surcoût dans la manipulation des objets. Il faudra donc optimiser le code pour éviter d'accéder au descripteur de classe à chaque accès à la variable.

Pour organiser les méthodes il est également nécessaire de prendre en compte le graphe d'héritage. Une méthode de coloriage permet également d'associer un décalage unique à chaque méthode. Cependant cette méthode pose le problème de l'extensibilité. Si du code est chargé dynamiquement alors il faut possiblement recalculer un coloriage et réorganiser les données.

Pour éviter ce problème, on peut utiliser une table de hachage pour associer à chaque champ un décalage unique.

7.2.3 Tester l'appartenance à une classe

Certains langages permettent de tester l'appartenance d'un objet à une classe. Cela peut se faire en suivant les liens pères dans les descripteurs de classe (en supposant de l'héritage simple). On peut également garder dans chaque descripteur de classe un tableau des classes ancêtre. Un objet x sera une instance d'une classe C si C est de niveau j dans la hiérarchie des classes (avec `Object` de niveau 0) et si dans le tableau de x, la j-ème valeur contient le descripteur de C.

[1]Le langage Java n'autorise pas l'héritage multiple, à la différence de C++.

Certains langages autorisent à restreindre la classe d'un objet. Dans un langage comme Java, cette restriction est accompagnée d'un test dynamique alors que dans C++, aucune vérification n'est faite ce qui peut conduire à des comportements imprévisibles.

7.2.4 Optimisations

Le coût le plus important dans un langage objet est l'appel de méthode. Il est essentiel d'analyser le code pour remplacer les appels dynamiques par des appels statiques. Par exemple, si une méthode n'est pas redéfinie dans une sous-classe alors il n'y a qu'un seul code possible à exécuter.

Chapitre 8

Ce qu'il faut retenir

Ce cours a présenté différents aspects de la compilation des langages de programmation. Parmi les points abordés, voici un court résumé des points à retenir :

- La distinction entre ce qui peut être traité par l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.
- La correspondance entre les langages reconnus et les expressions régulières et les grammaires.
- Le fonctionnement de l'analyse descendante et de l'analyse ascendante (en particulier la notion de lecture, réduction et les conflits qui en découlent).
- Le calcul des tables d'analyse descendante dans le cas LL(1) et des tables d'analyse ascendante dans le cas SLR(1).
- Le rôle des règles de précédence dans la résolution des conflits.
- La classification des différentes grammaires : LL(1), LR(0), SLR(1), LALR(1), LR(1), déterministes, ambiguës.
- La description d'un arbre de syntaxe abstraite.
- L'analyse de portée (la construction et la représentation de la table des symboles).
- Le rôle du typage, les règles de typage simple.
- Les constructions de base d'une machine à pile.
- La compilation des procédures à l'aide de tableaux d'activation.
- La représentation en machine de données complexes.

Chapitre 9

Description de la machine virtuelle utilisée en cours, TD et projet

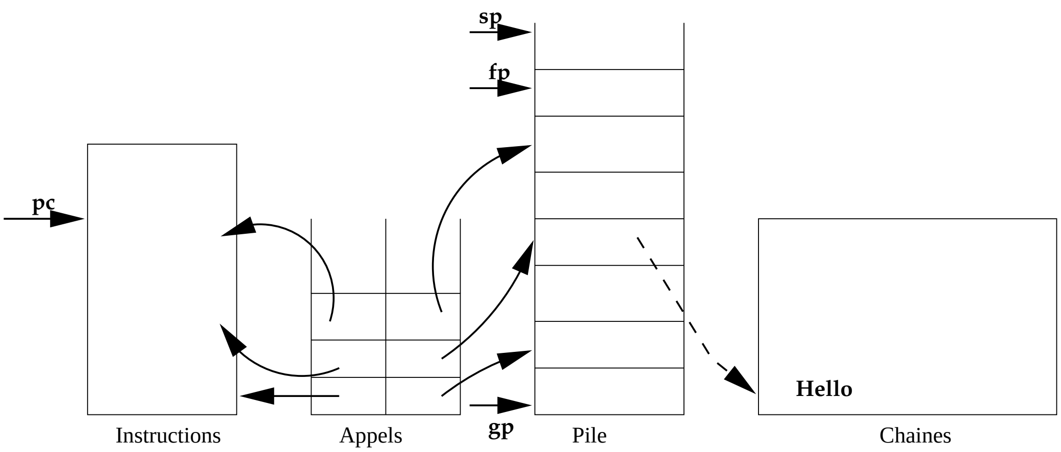
9.1 Description

Le code intermédiaire est exécuté par une machine à pile. La pile d'exécution contient des entiers, des réels et des adresses. Les chaînes de caractères sont stockées dans une mémoire séparée et repérées par leurs adresses. Trois registres permettent d'accéder à différentes parties de la pile.

- Le registre *sp* (stack pointer) repère le sommet courant de la pile, il pointe sur la première case libre de la pile.
- Le registre *fp* (frame pointer) repère l'adresse de base des variables locales.
- Le registre *gp* contient l'adresse de base des variables globales.

La machine comporte un registre *pc* qui pointe sur l'instruction courante du code à exécuter.

Une pile annexe permet de sauvegarder les appels, elle contient des couples de pointeurs, l'un sauvegardant le registre d'instructions *pc* et l'autre le registre *fp* des variables locales.



9.2 Instructions

Les instructions sont désignées par un nom et peuvent prendre un ou deux arguments. Ceux-ci peuvent être :

- des constantes entières,
- des constantes réelles,
- des chaînes de caractères entre double quote avec les mêmes conventions que le langage *C* pour ce qui concerne les caractères spéciaux `\"`, `\n`, `\\`,

- des adresses référençant des emplacements dans la zone des chaînes, dans la pile principale ou dans la pile d'appels,
- une étiquette désignant un emplacement dans les instructions soit sous forme d'un entier (on référence alors la n-ème instruction à partir du début du programme) soit une étiquette symbolique.

9.2.1 Terminologie

On utilisera les conventions suivantes :
- Empiler une valeur x signifie stocker x à l'emplacement $\mathsf{P}[sp]$ et incrémenter sp de 1.
- Empiler n fois une valeur x signifie itérer n fois l'opération précédente.
- Dépiler n valeurs revient à décrémenter de n la valeur de sp.
- Le sommet de la pile représente la dernière valeur stockée dans la pile soit $\mathsf{P}[sp-1]$, le sous-somment de la pile représente l'avant dernière valeur stockée dans la pile soit $\mathsf{P}[sp–2]$.

9.2.2 Opérations de base

Les opérations arithmétiques ou flottantes s'effectuent entre le sommet et le sous-sommet de la pile, les deux arguments sont dépilés et le résultat est empilé à la place. Le résultat des opérations de comparaison est un entier qui vaut 0 ou 1. L'entier 0 représente la valeur booléenne "faux" tandis que 1 représente la valeur booléenne "vrai".

Opérations sur les entiers

Instruction	Description
ADD	dépile n puis m qui doivent être entiers et empile le résultat $m+n$
SUB	dépile n puis m qui doivent être entiers et empile le résultat $m-n$
MUL	dépile n puis m qui doivent être entiers et empile le résultat $m \times n$
DIV	dépile n puis m qui doivent être entiers et empile le résultat m/n
NOT	dépile n qui doit être un entier et empile le résultat de $n=0$
INF	dépile n puis m qui doivent être entiers et empile le résultat $m<n$
INFEQ	dépile n puis m qui doivent être entiers et empile le résultat $m \leq n$
SUP	dépile n puis m qui doivent être entiers et empile le résultat $m>n$
SUPEQ	dépile n puis m qui doivent être entiers et empile le résultat $m \geq n$

Opérations sur les réels

Instruction	Description
FADD	dépile n puis m qui doivent être réels et empile le résultat $m+n$
FSUB	dépile n puis m qui doivent être réels et empile le résultat $m-n$
FMUL	dépile n puis m qui doivent être réels et empile le résultat $m \times n$
FDIV	dépile n puis m qui doivent être réels et empile le résultat m/n
FINF	dépile n puis m qui doivent être réels et empile le résultat $m<n$
FINFEQ	dépile n puis m qui doivent être réels et empile le résultat $m \leq n$
FSUP	dépile n puis m qui doivent être réels et empile le résultat $m>n$
FSUPEQ	dépile n puis m qui doivent être réels et empile le résultat $m \geq n$

Opérations sur les chaînes

Instruction	Description
CONCAT	dépile n puis m qui doivent être des adresses de chaîne, empile l'adresse d'une chaîne égale à la concaténation de la chaîne d'adresse n et de celle d'adresse m.

Égalité
Le test d'égalité teste que les deux objets (entiers, réels ou adresses) sur la pile sont égaux, Une erreur d'exécution a lieu si les deux objets ne sont pas de même type. Deux chaînes égales sont stockées à la même adresse, cette instruction permet donc de tester de l'égalité de deux chaînes.

Instruction	Description
EQUAL	dépile n puis m qui doivent être de même type et empile le résultat de $n = m$

Conversions
Différentes instructions permettent de convertir une chaîne de caractères en entier ou réel et réciproquement.

Instruction	Description
ATOI	dépile l'adresse d'une chaîne et empile sa conversion en nombre entier, échoue si la chaîne ne représente pas un entier.
ATOF	dépile l'adresse d'une chaîne et empile sa conversion en nombre réel, échoue si la chaîne ne représente pas un réel.
STRI	dépile un entier et empile l'adresse d'une chaîne représentant cet entier
STRF	dépile un réel et empile l'adresse d'une chaîne représentant ce réel

9.2.3 Manipuler des données

Si x désigne une adresse dans la pile alors $x[n]$ désigne une adresse située n cases au-dessus.

Empiler

Instruction	Argument	Description
PUSHI	n entier	empile n
PUSHN	n entier	empile n fois la valeur entière 0
PUSHF	n réel	empile n
PUSHS	n chaîne	stocke n dans la zone des chaînes et empile l'adresse
PUSHG	n entier	empile la valeur située en $gp[n]$
PUSHL	n entier	empile la valeur située en $fp[n]$
PUSHSP		empile la valeur du registre sp
PUSHFP		empile la valeur du registre fp
PUSHGP		empile la valeur du registre gp
LOAD	n entier	dépile une adresse a et empile la valeur dans la pile située en $a[n]$
LOADN		dépile un entier n, une adresse a et empile la valeur dans la pile située en $a[n]$
DUPN	n entier	duplique et empile les n valeurs en sommet de pile

Dépiler

Instruction	Argument	Description
POPN	n entier	dépile n valeurs dans la pile

Stocker

Instruction	Argument	Description
STOREL	n entier	dépile une valeur et la stocke dans la pile en fp[n]
STOREG	n entier	dépile une valeur et la stocke dans la pile en gp[n]
STORE	n entier	dépile une adresse a et une valeur v, stocke v à l'adresse $a[n]$ dans la pile
STOREN		dépile un entier n, une adresse a et une valeur v, stocke v à l'adresse $a[n]$ dans la pile

Divers

Instruction	Argument(s)	Description
CHECK	n, p entiers	vérifie que le sommet de la pile est un entier i tel que $n \leq i \leq p$, sinon déclenche une erreur
SWAP		dépile n puis m et rempile n puis m

9.2.4 Entrées-Sorties

Les instructions suivantes permettent de gérer les entrées-sorties.

Instruction	Description
WRITEI	dépile un entier et l'imprime sur la sortie standard
WRITEF	dépile un réel et l'imprime sur la sortie standard
WRITES	dépile l'adresse d'une chaîne et imprime la chaîne correspondante sur la sortie standard
READ	lit une chaîne de caractères au clavier, terminée par un retour-chariot, stocke la chaîne (sans le retour-chariot) et empile l'adresse.

9.2.5 Opérations de contrôle

Après l'exécution d'une instruction le registre *pc* est habituellement incrémenté de 1. Les opérations de contrôle permettent de modifier ce comportement par défaut.

Modification du registre *pc*

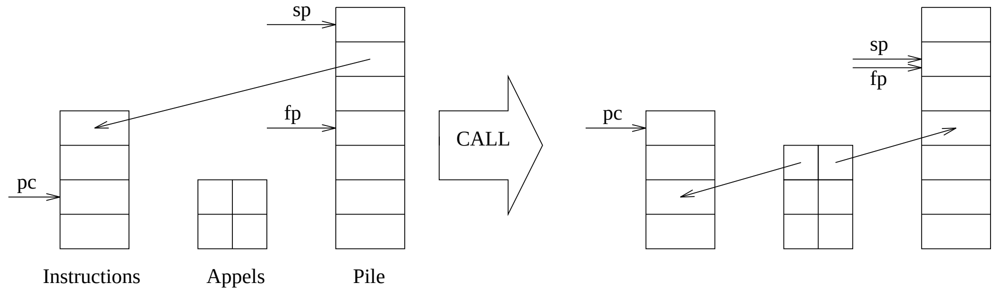
Instruction	Argument	Description
JUMP	*label* étiquette	affecte au registre *pc* l'adresse dans le programme correspondant à *label* qui peut être un entier ou une valeur symbolique.
JZ	*label* étiquette	dépile une valeur, si elle est nulle affecte au registre *pc* l'adresse dans le programme correspondant à *label* sinon incrémente *pc* de 1.
PUSHA	*label* étiquette	empile l'adresse dans le code correspondant à l'étiquette *label*

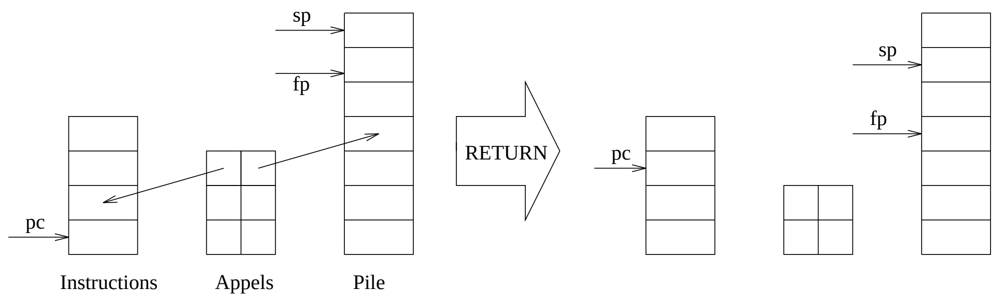
Procédure

Lors de l'appel de procédure il est nécessaire de sauvegarder les registre d'instructions et de variables locales qui seront restaurées au moment du retour.

Instruction	Description
CALL	dépile une adresse de code a, sauvegarde *pc* et *fp* dans la pile des appels, affecte à *fp* la valeur courante de *sp* et à *pc* la valeur a.
RETURN	affecte à *sp* la valeur courante de *fp*, restaure de la pile des appels les valeurs de *fp* et *pc* et incrémente *pc* de 1 pour se retrouver à l'instruction suivant celle d'appel.

Le comportement des instructions CALL et RETURN est décrit sur les schémas suivants où seuls les registres modifiés ont été dessinés :





9.2.6 Initialisation et fin

Dans l'état initial, le registre *pc* pointe sur la première instruction du programme. La pile des appels et la pile d'exécution sont vides. Les registre *gp* et *sp* pointent sur la base de la pile d'exécution tandis que *fp* n'est pas défini. Le registre *fp* doit être initialisé par l'instruction START qui ne peut être utilisée qu'une seule fois. Les instructions suivantes servent à stopper la machine à la fin du programme ou en cas d'erreur.

Instruction	Argument	Description
START		Affecte la valeur de *sp* à *fp*
NOP		ne fait rien.
ERR	x chaîne	déclenche une erreur d'instruction avec le message x.
STOP		arrête l'exécution du programme

Bibliographie

[1] A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers : principles, techniques and tools.* Addison Wesley, 1986.

[2] Claude Marché amd Ralf Treinen. Formation au langage caml. Université Paris Sud. Notes du Cours LGL-Deug MIAS.

[3] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[4] J.M. Autebert. *Langages algébriques.* Études et Recherches en Informatique. Masson, 1987.

[5] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Obective Caml.* O'Reilly, 2000.

[6] G. Cousineau and M. Mauny. *Approche fonctionnelle de la programmation.* Ediscience International, 1995.

[7] X. Leroy and P. Weis. *Manuel de Référence du Langage Caml.* iia. InterEditions, 1993.

[8] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc.* Unix Programming Tools. O' Reilly, 1995.

[9] J. Mitchell. *Foundations for Programming Languages.* Foundations of Computing. The MIT Press, 1996.

[10] P. Weis and X. Leroy. *Le Langage Caml.* iia. InterEditions, 1993.

[11] R. Wilhelm and D. Maurer. *Les compilateurs : théorie, construction, génération.* Manuels Informatique. Masson, 1994.