Compilation

L3 informatique

Université de Bordeaux

Vincent Penelle

2023-2024

Table des matières

Chapitre 1

Introduction

Préambule à la version 2024 du cours : ce poly consiste en des notes écrites peu avant de donner le cours en 2023, et légèrement mis à jour depuis. Comme c'est la seconde année que je donne ce cours, il est en version beta. À ce titre, il est probablement incomplet et changera sans doute encore pas mal. Donc ne l'imprimez pas et consultez plutôt la version en ligne.

1.1 Généralités et but du cours

Dans tous les langages que vous avez manipulé depuis que vous programmez, vous avez utilisé des compilateurs (pour le C, le Java, le OCaml, etc), ou des interpréteurs (en Python, en OCaml, en bash, en html, etc). Jusqu'ici, vous avez très probablement considéré ces outils comme des boîtes noires vous permettant d'exécuter votre code. Dans le cadre de ce cours, nous allons (partiellement) ouvrir cette boîte pour voir un peu comment c'est fait, quels outils et techniques un compilateur ou un interpréteur utilise, et ce qu'ils peuvent et ne peuvent pas faire. Pour cela, on en implémentera un, pour un langage minimal bien évidemment (c'est déjà suffisamment compliqué).

Les buts principaux de cet enseignement sont les suivants :

— Certaines techniques utilisées en compilation, notamment le *parsing* (mais pas que) sont en réalité utiles en dehors de la compilation. Le *parsing* vous sera par exemple utile dès que vous aurez à écrire des programmes devant analyser des données un peu structurées au format texte, sans avoir nécessairement à les faire entrer au forceps dans des formats tout faits (tels que le `json` qui dispose d'un parseur générique).

— Un programme informatique est un objet difficile à définir. On passera une partie du cours à définir proprement ce qu'est sa *sémantique* ; c'est-à-dire son effet sur l'état d'un ordinateur, et en particulier de sa mémoire. C'est même tellement important qu'on commencera par là.

— Avec l'analyse sémantique, on regardera quelles informations un compilateur est capable (ou pas) d'analyser sur un programme, comment il le fait et comment il peut en informer l'utilisateur. Cela devrait vous aider à mieux comprendre les messages d'erreur que vous renvoie le compilateur.

— Les compilateurs réalisent des optimisations sur le code qu'ils génèrent, en simplifiant des redondances dans les calculs, calculant directement les constantes, etc. On en explorera certaines pour vous donner une meilleure idée de ce que fait votre compilateur favori de votre code, et ainsi, vous aider à mieux l'optimiser en ne le sur-optimisant pas (il est inutile de rendre votre code moins lisible par une optimisation que le compilateur fera de toutes manières). Si vous voulez être un bon

programmeur, il est indispensable que vous maîtrisiez le (ou les) langages dans lesquels vous développez, et les performances de vos programmes seront intimement liées à ce qu'en fait le compilateur ou l'interpréteur. Ainsi, vous ne pourrez être spécialiste d'un langage sans bien connaître ces outils.
— Les problèmes résolus par les compilateurs ne sont pas triviaux, et cela nous permettra aussi d'illustrer à quoi sert la théorie, puisque la plupart de ces problèmes se voient résolus par des techniques développées par la théorie (par exemple, le parsing bénéficie énormément de la théorie des langages).
— Enfin, l'implémentation sera faite en OCaml, ce qui nous permettra d'illustrer ce langage sur un type de problème où il est très bien adapté, et d'avoir l'occasion de l'utiliser sur un projet conséquent (notez que ceci n'est tout de même pas un cours de OCaml).

Je liste tout cela dès le départ pour être clair sur un point : bien évidemment, il est peu probable que vous écriviez des compilateurs en-dehors d'un tel cours dans un cadre professionnel, et de toute manières un cours d'introduction ne saurait vous donner toutes les techniques modernes pour ce faire (et d'ailleurs ce cours ne présentera que peu les problématique modernes de compilation, pour rester sur celles, fondamentales qui existaient déjà dans les années 80-90). Mais ce n'est pas grave du tout : le but d'un tel cours est de vous donner des outils pour mieux utiliser un compilateur et mieux développer.

1.2 La compilation et l'interprétation

Vous le savez déjà (vous l'avez abordé en cours d'Architecture des Ordinateurs), un ordinateur ne sait pas lire directement de code dans un langage de programmation. Le processeur est uniquement capable d'exécuter un nombre limité d'instructions sur des valeurs stockées dans un nombre limité de registres et d'accéder à la mémoire en y copiant des informations vers ou depuis les registres. Et c'est tout. D'ailleurs un point contre-intuitif de tout ça, c'est que la liste d'instructions exécutée est elle-même codée dans une partie de la mémoire auquel le processeur accède via un registre particulier et que rien ne distingue a priori des données. Il est même tout à fait possible que l'exécution d'un programme modifie son propre code (ce qui est plus que déconseillé).

Il y a là un contraste tout à fait frappant avec les langages modernes qui interdisent par défaut ce dernier comportement, disposent d'une mémoire où on peut accéder directement à toutes les valeurs et disposent d'instructions variées et de structures de données complexes.

Or puisque l'ordinateur ne peut pas lire directement de tels langages, il faut bien traduire ces langages vers des codes lisibles par le processeur. Et en très gros, c'est ce que fait un compilateur.

Au passage, l'assembleur (ou plutôt les assembleurs) sont déjà des langages qui se compilent vers du langage machine (ou plutôt s'assemblent, puisque la tâche est nettement moins complexe), et il y a divers programmes qui réalisent ces tâches, avec des syntaxes différentes (`as`, `nasm`, etc), ainsi que l'éditeur de liens (`ld`) qui lie des bouts de programmes compilés pour que les bonnes portions de codes soient appelées par les appels de fonction. Bon, ces langages d'assembleurs sont en réalité essentiellement une version lisible par un humain du code machine (à l'édition des liens près) et sont donc dépendants de la machine où ils s'exécutent (puisque le jeu d'instruction dépend du processeur, ce n'est donc pas le même sur des processeurs différents). On ne s'étendra donc pas dessus, la partie traduction vers du langage machine étant une traduction assez directe (donc ça n'ajouterai rien au propos), et la partie édition des liens étant à la fois peu complexe et hors de propos.

D'une manière plus générale, un compilateur n'a pas nécessairement pour cible un

langage machine. Certains langages sont compilés vers du C (qui a des compilateurs assez efficaces). On peut donner la version très générale d'un compilateur suivante :

Définition 1.2.1 (Compilateur)**.** *Un compilateur est un programme qui transforme un fichier texte décrivant un programme dans un langage L_1 dans un fichier texte décrivant le même programme dans un langage L_2.*

C'est très légèrement incorrect lorsque la cible est du code machine (il manque l'édition des liens et 2-3 autres trucs), mais pour ce qu'on verra ensemble, ce sera tout à fait adapté et ça reste très proche de la réalité.

On a parlé d'interprétation un peu plus haut, il est temps également de définir cela un peu mieux :

Définition 1.2.2 (Interpréteur)**.** *Un interpréteur est un programme qui simule l'exécution d'un programme P écrit dans un langage L.*

Évidemment, l'interpréteur est un programme qui a, lui, été compilé sur la machine où il s'exécute (enfin, on peut faire un interpréteur interprété dans un autre interpréteur, mais ultimement il y aura une machine réelle qui exécute, et c'est pas méga-malin d'empiler des interpréteurs).

Les avantages de l'interprétation est que le travail à réaliser sur le code est moins important (puisque l'interpréteur est capable de comprendre des instructions de plus haut niveau) et qu'il n'y a pas à tenir compte des spécificités de la machine physique où l'interpréteur est exécuté (puisque c'est l'interpréteur qui en dépend). Cela rend donc un code portable.

Les inconvénients sont que le programme doit être parsé et interprété à chaque exécution, et que pour que le temps de démarrage ne soit pas trop long, on ne peut pas y pratiquer autant d'optimisations que pour un code compilé. Les programmes interprétés ont donc tendance à être moins efficaces que leurs équivalents compilés.

Il convient de nuancer le point précédent : certains langages (Python par exemple) sont conçus pour être interprétés et sont donc directement optimisés pour cela. Il existe maintenant des compilateurs Python, mais le langage étant conçus pour être interpréter, ils sont moins efficaces que des compilateurs pour des langages dont la conception prévoit la compilation. Les langages de scripts, par exemple le `bash` ou le `perl`, mais également le `html`, le `javascript` ou le `php` sont également conçus pour être interprétés (pour le `html` et le `javascript`, par un navigateur web, pour le `php`, par le serveur), et leur but rend leur compilation peu utile : en effet, le but de ces langages est justement d'être portables (pour les trois derniers), ou «jetables» (pour les deux premiers) et on perdrait donc à la compilation (vous ne voulez pas qu'une page web soit compilée chez vous avant de s'exécuter).

Il y a également une approche intermédiaire entre les deux, choisie notamment par Pascal, Java ou OCaml (ce dernier ne choisissant pas que cela) qui consiste à faire une passe de compilation complète, mais d'avoir en sortie un bytecode d'une machine virtuelle bas niveau (similaire à de l'assembleur, mais avec une gestion de la mémoire plus souple) et d'exécuter ce bytecode sur un interpréteur (qu'on appelle machine virtuelle). On gagne alors un peu des deux approches, le bytecode pouvant être optimisé lors de la phase de compilation et étant suffisamment bas niveau pour que son interprétation soit relativement efficace, et en produisant un bytecode portable qui peut être partagé facilement, et où le debbugage est plus aisé que sur un binaire. Toutefois, pour un langage comme OCaml où la compilation peut produire un binaire ou du bytecode, on peut observer que le binaire tend à être plus rapide à l'exécution.

1.3 Rapide remarque historique

On l'a vu plus haut, un compilateur est un programme qui transforme des programmes depuis un langage source vers un langage cible, sans en changer la sémantique. On a vu également que de base, un ordinateur est simplement capable d'exécuter les instructions de son processeur en lisant un code binaire qui y correspond. Pourtant, la plupart des compilateurs sont écrits dans des langages haut niveau, voire dans le langage qu'ils compilent.

Ça pose donc une question assez perturbante : comment on a fait au début ? En effet, comment pourrait-on compiler un compilateur avec lui-même s'il n'a pas déjà été compilé ? La réponse n'est pas si surprenante : le premier compilateur d'un langage ne peut pas être écrit dans ce langage, mais dans un autre préexistant, et éventuellement sur une faible partie du langage, qui servira d'amorçage avant de compiler un compilateur plus complet écrit dans le langage cible et compilé avec ce premier compilateur.

Ce problème s'appelle le problème d'amorçage ou *bootstrap* en anglais, et est en réalité même plus général. Le tout premier assembleur a évidemment été écrit directement en binaire.

Avant cela (dans les années 1940), on traduisait à la main des programmes en codes binaires que l'on câblait directement sur la machine à l'image des réseaux téléphoniques, ce qui évidemment provoquait de nombreuses erreurs. Au milieu des années 1950, on a vu l'apparition des premiers vrais langages d'assembleurs (le premier sur l'IBM 701) avec une traduction automatique vers le langage machine associé. À ce stade, la compilation d'assembleur est simplement une traduction directe en binaire d'instruction assembleurs, mais le premier d'entre eux a bien été écrit directement en binaire !

Assez rapidement après cela (à la fin des années 1950), on a vu apparaître les premiers langages de programmation (A-0 pour l'arithmétique, Fortran pour les calculs numériques et la programmation générique, COBOL pour les traitements de gestion, Algol 58 pour la traduction des algorithmes en langage de haut niveau), dont les premiers compilateurs furent écrits en assembleur.

Puis, dans les années 1960, LISP a été le premier langage à disposer d'un compilateur écrit en LISP (mais évidemment, ce n'était pas le premier compilateur LISP).

À partir des années 1970, il est devenu courant d'avoir des compilateurs écrits dans le langage qu'ils compilent, et c'est le cas de la majorité des langages généraux que vous connaissez, du moins à terme, puisqu'il y a toujours le problème d'amorçage. D'ailleurs, ce problème peut poser des bugs, puisque un bug d'un programme peut très bien venir d'un compilateur buggué, et donc un compilateur dont on arriverait à démontrer la correction du code pourrait très bien produire un exécutable buggué si le compilateur qu'on utilise pour le compiler l'est. C'est un problème difficile à résoudre qui nécessite d'être très attentif et prudent lorsque l'on s'attache à cette tâche.

Pour aider à tout cela, toujours dans les années 1970 sont apparus des programmes simplifiant l'écriture des compilateurs, à savoir les compilateurs de compilateurs (ou *compiler-compiler*) qui permettent de décrire par un méta-langage le langage analysé et traduit par le compilateur.

1.4 Le fonctionnement d'un compilateur en (très) bref

Bon, mais comment fonctionne un compilateur ?

[À terme, faire un dessin]

Un compilateur découpe son travail en plusieurs étapes relativement simples mais assez techniques qui permettent d'avoir des modules possibles à maintenir. Ces étapes sont géné-

ralement découpées en deux blocs qu'on appelle respectivement le *front-end* et le *back-end*. Le *front-end* regroupe toutes les étapes qui traitent le langage indépendamment de la machine où il va finalement s'exécuter, et qui devront aussi être réalisées par un interpréteur (on y reviendra), et le *back-end* regroupe toutes les étapes qui visent à la production du code sur la machine cible et à son optimisation au niveau de cette machine. Ce découpage n'est pas du tout arbitraire. Il repose sur le fait que les deux blocs vont interagir via un langage intermédiaire bien choisi, qui permettrait théoriquement de développer les deux indépendamment, et ainsi d'interfacer plusieurs *back-end* sur un même *front-end* et vice-versa. En réalité, c'est déjà ce que font la plupart des compilateurs (par exemple, gcc a une version pour arm, ce qui est assez différent d'intel, et aura le même *front-end* sur les deux versions). C'est d'ailleurs là tout le sens d'un projet tel que LLVM qui visait à l'origine à fournir un langage intermédiaire universel pour décharger les producteurs de compilateurs de la tâche d'écrire un *back-end*.

Les phases du *front-end* sont les suivantes :

— Analyse lexicale (*tokenization*) : traduire le fichier source en une séquence de tokens sur laquelle s'exécutera l'analyse syntaxique (pour la faciliter).
— Analyse syntaxique (*parsing*) : traduire la séquence de tokens en un arbre de syntaxe abstraite (AST) qui représente le programme sous une forme arborescente pour en faciliter l'analyse.
— Analyse sémantique : analyser l'AST pour y collecter un certain nombre d'informations, détecter des erreurs potentielles, et effectuer des simplifications et optimisations. Typiquement, à cette phase vont se situer le typage statique pour les langage qui en disposent (vérification de type ou inférence, selon les cas), l'analyse de la portée des variables, simplification des termes constants.
— Production de code intermédiaire : transformer l'AST en une séquence d'instructions dans un langage intermédiaire bien choisi et assez proche de l'assembleur (typiquement, du code à 3 adresses), tout en conservant sa sémantique. Cette dernière phase est en réalité un peu intermédiaire entre les deux blocs, et on pourrait très bien faire le choix d'un langage intermédiaire proche de l'AST et avoir cette phase dans le *back-end*.

Les phases du *back-end* sont les suivantes :

— Une phase d'optimisation sur le code à 3 adresses, qui se fait généralement en le découpant en blocs d'instructions sans sauts, en voyant le résultat comme un graphe (de flot) et en optimisant chaque bloc et certaines instructions entre les blocs. Ceci va conduire à détecter des valeurs redondantes ou constantes, à éliminer du code mort, etc. Ces optimisations sont majoritairement capturées par ce que l'on appelle des algorithmes de flot de données.
— La sélection des instructions de la machine pour correspondre à nos instructions abstraites de notre code à 3 adresses (ce ne sont pas nécessairement les mêmes). Étant donné que notre cible sera de l'assembleur, ce point sera relativement simple.
— La sélection des registres : le code à 3 adresses a une infinité de noms de variables, il faut dans le code produit les répartir entre la mémoire et les registres (limités) disponibles. Des algorithmes de coloration de graphes sont utilisés pour cela.
— La détermination de l'action du ramasse-miettes (dans les langages qui en disposent) se fait à ce niveau. Ce point ne sera (probablement) pas abordé dans ce cours.
— Une dernière phase d'optimisations sur le code machine produit (pour gérer les redondances éventuellement ajoutées ici).
— L'écriture du code machine produit et l'édition des liens (encore une fois, puisqu'on produit de l'assembleur, on n'aura pas cela à traiter dans ce cours).

Les interpréteurs vont essentiellement s'arrêter au *front-end* (avant la traduction en code à 3 adresses) et directement interpréter les instructions (en simulant l'effet des instructions sur une abstraction de la machine).

Bon, alors le schéma ci-dessus correspond aux compilateurs tels qu'ils existaient au milieu des années 90. Ça nous suffira largement parce que c'est déjà beaucoup (on ne fera pas tout) et qu'en partant de zéro, il est difficile de faire vraiment différemment. Cependant dans les compilateurs modernes, il existe tout un tas de trucs en plus qui peuvent exister (et qui sont bien au-delà de ce cours) :

— Certains langages sont exécutés sur une machine virtuelle au lieu d'être compilés vers de l'assembleur, dans un intermédiaire assez souple entre compilation et interprétation. C'est notamment le cas de Pascal, Java ou d'OCaml (ce dernier peut également être compilé vers du code natif). Cela permet en particulier d'avoir des exécutable portables (seule la machine virtuelle étant spécifique à la machine), mais un peu plus lent, et également de débugguer les programmes plus facilement.

— Certaines machines virtuelles du cas précédent disposent également de la possibilité de faire du *just in time compilation*, c'est-à-dire de compiler à la volée des parties du code interprété pour en accélérer l'exécution. Dans de nombreux cas, cela rend l'efficacité des programmes assez proches d'une compilation directe. C'est le cas en Java, notamment.

— Le ramasse-miette peut être un programme indépendant qui s'interface alors avec le programme compilé. Dans le cadre d'un langage interprété (ou d'une machine abstraite), l'interpréteur ou la machine abstraite fera elle-même ce travail.

— Les processeurs modernes parallélisent certaines instructions, et un compilateur fortement optimisant doit tenir compte de ce fait dans la sélection et l'ordonnancement des instructions s'il veut rendre son programme le plus efficace possible. Cela donne lieu à des tas de compromis à effectuer avec les autres optimisations et donne lieu à des questions complexes totalement hors de notre portée.

— Dans certains compilateurs, on a plutôt 3 blocs plutôt que 2, avec, entre le *front-end* et le *back-end*, un *middle-end* qui va effectuer les optimisations de haut-niveau (ne laissant au *back-end* que les parties spécifiques à l'architecture cible) et également être capable de lier des bibliothèques potentiellement écrites dans d'autres langages. C'est grâce à ce genre de techniques qu'on peut par exemple appeler du code C depuis du OCaml ou du Python de manière relativement transparente.

— Enfin, il commence à exister des compilateurs certifiés qui sont capable de démontrer que le code machine produit a réellement la même sémantique que le code d'entrée, tout en effectuant un certain nombre d'optimisations. C'est notamment le cas de compcert qui est un compilateur C certifié, codé en grande partie en Coq et en OCaml (avec menhir).

1.5 Un mot sur la complexité des compilateurs/interpréteurs

On pourrait croire que les compilateurs et les interpréteurs, au vu des tâches complexes qu'ils ont à réaliser, vont avoir besoin d'algorithmes gourmands en temps et en mémoire. En effet, pour certains problèmes à résoudre, par exemple l'affectation des registres, on ne connaît pas d'algorithme exact s'exécutant en temps polynomial. Et encore, des algorithmes quadratiques sont déjà trop gourmands pour de nombreux cas. Et dans l'absolu on pourrait faire des compilateurs/interpréteurs qui utilisent des algorithmes gourmands. Néanmoins, en pratique un tel compilateur/interpréteur ne serait pas utilisable.

En effet, un interpréteur qui mettrait une minute entre chaque pas d'exécution sur de

long programme serait peu utile en pratique (et vous avez déjà râlé sur un script qui rame dans votre navigateur, donc imaginez la catastrophe si c'était intrinsèque à l'interpréteur). De même, des programmes un peu conséquents font facilement des dizaines de milliers de code. Un compilateur qui serait ne serait-ce que quadratique en la taille du code mettrait un temps déraisonnable à compiler de tels projets (et déjà, même avec de bons compilateurs et une machine raisonnable, compiler quelque chose comme un noyau linux prend en général des heures).

Les algorithmes qu'on utilisera dans un compilateur doivent donc être linéaires ou à la limite quasi-linéaires ($n\log(n)$) pour que le compilateur puisse être utilisable sur des projets un peu gros. Cela nous amènera dans certains cas (par exemple l'affectation de registres) à utiliser des algorithmes approximés mais avec une bonne complexité (tout en donnant un résultat très raisonnables). Vous aurez plus de détails sur ces techniques dans le cours de Techniques Algorithmiques et Programmation.

L'usage des compilateurs a évolué dans l'histoire de l'informatique. Il était naturel pour un programmeur des années 1970 d'avoir le résultat de la compilation de son programme le lendemain ou plus tard dans la semaine. Dans les années 1980, un petit programme de quelques centaines de lignes pouvait mettre une heure ou deux à compiler. Il était alors important d'obtenir un rapport des erreurs le plus complet et détaillé possible et il était même parfois utile d'utiliser des compilateurs sachant réparer automatiquement le code donné. Nous ne verrons pas cette pratique nommée *autoréparation* qui a été abondamment étudiée mais qui est devenue aujourd'hui obsolète. Mais malgré l'évolution des performances des machines, l'étude théorique de la complexité algorithmique en compilation est importante, car des algorithmes exponentiels, ou même quadratiques, restent rédhibitoires sur des données importantes.

1.6 Plan du cours

Le cours suivra un plan relativement atypique pour un cours de compilation, à savoir que nous ne respecterons pas totalement l'ordre des étapes d'un compilateur.

Nous commencerons par définir formellement ce qu'est un programme et ce qu'est sa sémantique, dans le chapitre 2. Cela nous permettra de fixer les idées sur ces concepts importants. On illustrera ces notions sur un langage simpliste (mais pas trop) fortement inspiré du Pascal, et on en implémentera un interpréteur. À ce stade, les programmes resteront des objets abstraits et on ne pourra donc pas les écrire en dehors du code (ce qui est peu pratique, mais volontaire).

Ensuite, on s'attaquera à régler ce dernier problème, et on consacrera un long chapitre 3 à l'analyse syntaxique, découpée en analyse lexicale (*tokenization*) et en analyse syntaxique (*parsing*) qui permettra donc de transformer un fichier texte en un programme au sens défini au chapitre 2. On en profitera pour discuter des algorithmes utilisés pour réaliser ces tâches, faire le lien avec les concepts théoriques (automates finis et grammaires algébriques que vous avez déjà croisés) et vous faire découvrir des outils, ocamllex et menhir qui les implémentent pour vous (ce ne sont évidemment pas les seuls). On les utilisera également pour d'autres tâches que de la compilation.

Au chapitre 4, on fera le point sur quelques analyses sémantiques qu'on peut effectuer sur un AST, en particulier le typage statique d'un programme, et comment on peut profiter de ces analyses pour écarter des programmes qui planteraient à l'exécution et indiquer des erreurs à l'utilisateur.

Au chapitre 5, on parlera de la transformation de l'AST en code à 3 adresses, puis de la transformation de ce dernier en code assembleur (relativement naïf). Si le temps le permet,

on parlera d'affectation de registres. À ce stade, on devrait avoir un compilateur complet (mais peu efficace et peu débuggable).

Enfin, au chapitre **??**, on parlera de l'optimisation que l'on peut faire sur le code à 3 adresses, en particulier d'algorithmes de flot de données.

1.7 Un mot sur la partie implémentation

Les parties TP de ce cours, ainsi que le projet (qui en sera très probablement une extension) seront effectuées en OCaml qui est un langage que vous avez déjà croisé.

Ce choix (pour le coup assez standard, la plupart des cours de compilation se répartissent entre C, C++, Java et OCaml ou un autre langage de la famille ml) n'est pas dû au hasard. En effet, comme vous l'avez sans doute compris, les arbres sont assez omniprésents dans un compilateur, et la facilité de les définir par des types et la souplesse offerte par le pattern-matching permettent de les manipuler assez aisément via un code relativement clair (comparé à ce qui serait le cas en C où la gestion de la mémoire ne serait pas triviale).

Par ailleurs, le générateur de parseur le plus récent en OCaml, menhir est assez puissant et intéressant : il réalise des parseurs LR(1) (contrairement à beaucoup de générateurs de parseurs, tels que yacc), ce qui permet d'exprimer des grammaires plus naturellement que dans d'autres cas, il a un retour d'erreur plus clair que la plupart des autres générateurs et fourni une représentation textuelle et graphique de l'automate généré (ce qui nous aidera pour comprendre ce qu'il y a derrière), et il a une API assez développée, ce qui m'a permis de développer un petit outil de visualisation d'exécution qui aurait été impossible sans cela.

Enfin, d'une manière plus prosaïque, cela vous donnera l'occasion de pratiquer un projet conséquent dans un langage fonctionnel et de voir un certain nombre d'outils qui y sont associés. En particulier, on utilisera des fonctions d'IDE qui faciliteront la vision de ce que vous êtes en train de faire, et vous pourrez tester des fonctions à la volée grâce au *toplevel*. Évidemment, le cours n'étant pas un cours de développement, vous ne ferez pas tout de zéro, et une bonne partie de la configuration et du code englobant vous seront fournis.

Chapitre 2

Définition d'un langage de programmation

2.1 Introduction

Avant d'attaquer la compilation proprement dite, on va définir le plus proprement (donc formellement possible) une notion de programme et sa sémantique : en effet, avant de décrire comment fonctionne un compilateur, il faut être capable de décrire les objets qu'il manipule. Cela va nécessiter de définir une machine abstraite (qui n'est pas un ordinateur complètement réaliste, mais pas si loin que ça), ainsi qu'un programme comme un objet mathématique (à savoir un arbre) et de définir l'effet de chaque instruction du programme sur la mémoire de la machine abstraite.

Il faut savoir que les langages les plus généraux sont rarement entièrement définis dans ces termes (parce que c'est compliqué et qu'historiquement ça n'est pas apparu comme ça), mais c'est un travail qui théoriquement pourrait être fait, et clarifierait pas mal de points sombres dans la sémantique réelle de certains langages (il arrive que des comportements soient mal définis, et donc que tous les compilateurs d'un langage ne fassent pas la même chose sur certains programmes, ce qui est gênant).

Cette définition formelle est utile puisqu'au final, la vraie définition de ce que fait un programme, c'est bel et bien son effet sur la mémoire de l'ordinateur. Et lorsqu'on écrit un compilateur, on doit s'assurer que l'effet du code produit correspond bien à l'effet attendu du code que l'on compile. Ce qui est évidemment délicat si le code d'entrée n'a pas d'effet bien défini.

Aussi le but de ce chapitre est de vous montrer ce que peut-être une sémantique bien définie – même si on verra en le faisant que même comme ça, ça reste difficile d'en déduire le comportement exact d'un code.

Pour cela, on ne va pas faire de chapitre trop générique, mais se concentrer sur un mini-langage impératif, inspiré partiellement de C (pour sa syntaxe) et de Pascal (pour la gestion des passages de paramètres et l'absence de pointeurs explicites). À ce titre, on ne considérera pas toutes les constructions et leurs subtilités qui peuvent se présenter dans un langage de programmation et dans sa compilation. Par exemple, on n'aura ni objets, ni fonctions de première classe, ni fonctions génériques, ni clôtures, mais uniquement un mini-langage impératif. Ce choix est là pour que l'on soit capable d'écrire un compilateur complet à notre niveau. Évidemment, cela veut dire qu'on n'abordera pas de nombreux concepts dans notre cours, mais c'est le prix à payer pour conserver une certaine clarté.

On va donc dans un premier temps définir une machine abstraite, ainsi que sa mémoire (qui sera manipulée à travers un environnement), qui servira de modèle d'exécution à notre

langage, puis une syntaxe pour notre langage (sous forme d'une structure arborescente), et enfin une sémantique à ce langage qui sera donc un fonction prenant en entrée un état de mémoire de la machine et renvoyant l'état de la mémoire modifié.

Ce chapitre est très lié au TD machine correspondant qui consiste en l'implémentation d'un interpréteur du langage présenté ici : le TD consistera à implémenter le module `Interpreter`. La machine abstraite définie ci-après vous sera donnée dans le module `Abstract_machine` (elle est vue comme une boîte noire en fonction de laquelle la sémantique est définie), et vous aurez à implémenter les fonctions du module `Interpreter` qui correspondent à la sémantique du langage présentée dans ce chapitre.

2.2 Machine abstraite

On commence par définir la machine abstraite qui va exécuter notre programme. C'est un modèle de calcul simple qui contient une mémoire (a priori infinie) pouvant stocker des données de différents types et d'opérations de base qu'elle peut exécuter sur ces données. Cette machine peut être vue comme un ordinateur idéalisé, et est plus simple à décrire qu'un ordinateur réel : si la gestion de la mémoire est très simplifiée (essentiellement la mémoire est simplement indexée par des chaînes de caractères, via des environnements qui sont des espaces de noms, et les types de données restent distincts), les seules opérations de base utilisables sur les données sont celles de l'assembleur. Elle servira essentiellement à définir simplement la sémantique de notre langage, ce qui sera utile pour générer du code bas niveau quand on cherchera à compiler vers de l'assembleur, puisque ce code devra respecter la sémantique définie ici. L'intérêt de ce genre de définition est donc d'être précis sur les opérations du langage tout en n'ayant pas à être trop précis sur la gestion de la mémoire et la concrétisation du flot de contrôle – qui apparaîtront dans la sémantique, mais pas dans la machine elle-même. Évidemment, dans le passage vers l'assembleur, il faudra traiter ces questions.

Notre machine est capable de manipuler les types de base suivants : `int`, `float` et `bool`. On ne précise pas plus leurs particularités, mais on considère que ce sont les mêmes objets que dans un ordinateur standard. On pourrait cependant considérer une machine manipulant de vrais nombres mathématiques sans changer aucunement la définition de la sémantique qui suivra (mais ce serait évidemment moins simple à implémenter, et ce serait moins utile pour un langage qui a vocation à être compilé). Elle dispose également d'un type de donnée représentant un pointeur – qui sera utiliser pour représenter les tableaux (notre langage ne permettra pas de manipuler autre chose et n'aura notamment aucune arithmétique des pointeurs).

Formellement, on a le types de données suivants :

```
value = int | float | bool | array | undef
```

Dans le code OCaml, ils seront stockés dans un type somme, qui permettra de contenir les données associées à la valeur. Chaque type de base sera donc un constructeur avec une donnée, à part `undef` qui ne contient pas de donnée, et `array` qui en contiendra deux (un nom et un environnement, on détaille cela plus loin).

Notre machine dispose d'opérations sur les éléments de `value` qui sont de type `value` $\rightarrow$ `value` $\rightarrow$ `value` et qui sont des fonctions partielles : $+_i, -_i, \times_i, /_i, \%_i$ qui effectuent les opérations usuelles sur les `int` et ne sont pas définies si les deux arguments ne sont pas des `int`. Par exemple, formellement on a : $a +_i b = \begin{cases} a + b \text{ si a, b et a+b sont des } \texttt{int} \\ \text{non-défini sinon} \end{cases}$, où $+$

est l'addition usuelle sur les `int` (i.e., l'addition normale quand le résultat mathématique est dans l'intervalle des `int` (`[INT_MIN;INT_MAX]`), et réalise un overflow sinon (valeur non-définie, dépendant de l'architecture)). Elle dispose de même d'opérations similaires sur les flottants : $+_f, -_f, \times_f, /_f, \%_f$, d'opérations sur les booléens $\wedge_b, \vee_b, \neg_b$ (le dernier étant une opération unaire) et de comparaisons $=_m, <_m$ qui sont capables de comparer deux `value` de même type. Ces opérations sont des primitives de notre machine. On pourrait se contenter de ne pas leur donner de sémantique précise, puisque la sémantique de notre langage sera définie en fonction de ces opérations. Néanmoins, puisqu'on devra ensuite produire du code assembleur de même sémantique, on va exprimer leur sémantique suffisamment précisément de manière à ce qu'elle corresponde à des opérations assembleur (ou qu'elle soit facilement traduisible dans des petits codes assembleurs). Cependant cette sémantique n'est pas vraiment formelle puisqu'elle sera partiellement exprimée en français et qu'on y laisse sous silence les particularités de limites de représentations de la machine (pour les `int`, les résultats non-définis à cause des overflow, pour les `float` les résultats non-définis et les problèmes d'arrondis). On résume les opérations existantes, ainsi que leur sémantique et la correspondance avec les fonctions du module `Abstract_machine` du code de l'interpréteur que vous produirez (le module correspondant vous sera donné), en figure 2.1 et 2.2.

À noter que lorsque le résultat d'une opération sur deux valeurs n'est pas défini, cela signifie soit que la machine plante (ce qui sera notre cas ici), soit qu'il peut se passer n'importe quoi (ce qui est par exemple le cas en C, par exemple dans la lecture d'une zone mémoire non initialisée). En théorie, dans ces cas-là, aucune sémantique n'existe (le cas n'existe pas), néanmoins, en pratique, si de tels calculs sont lancés, il se passe quelque chose et concrètement, quand aucune sémantique n'est donnée, cela veut dire qu'on ne garantie rien du tout. Dans cette partie, le langage pourra définir des programmes provoquant ces comportements. On verra comment, dans les autres passes de compilation, rejeter des programmes qui les provoqueraient.

La mémoire de la machine est considérée comme étant un tableau infini dont chaque case peut contenir une `value`.

On considère Var un ensemble (potentiellement infini) de variables. Dans la compilation par la suite, ces variables correspondront à des identifiants qui seront constitués de caractères alphanumériques et d'underscore, mais devant commencer par une lettre.

L'accès à la mémoire se fait via un environnement ou valuation qui est une fonction de Var vers les cases de la mémoire. On dispose d'opérations qui permettent de lire et d'écrire dans ces cases mémoires.

Ainsi, étant donné un environnement ρ et une variable $x \in \text{Var}$, $\rho(x)$ désigne la valeur stockée dans la case pointée par x dans ρ, et $\rho[x \leftarrow n]$ est une instruction qui place n dans la case mémoire pointée par x dans ρ. Au final, même si elle n'apparaît pas explicitement par un symbole, l'opération $\rho[x \leftarrow n]$ modifie bien la mémoire, et non ρ lui-même (qui fait toujours pointer les mêmes variables vers les mêmes cases). Le but de cette convention est de simplifier un peu ce qu'on écrira dans la sémantique des instructions les plus simples de notre langage. Quand on manipulera des fonctions et des tableaux (et donc qu'on aura besoin de plusieurs environnements), on explicitera ce rôle pour clarifier.

Dans ce cas, il faudra représenter la mémoire de l'ordinateur et sa vue par une fonction comme un couple M, ρ, où M est la mémoire, et ρ l'environnement qui est simplement ici une fonction des noms vers des cases mémoires de M. Dans ce cas, la mise à jour d'une variable x dans l'environnement ρ pourra s'écrire $(M[\rho(x) \leftarrow n], \rho)$.

Un point à préciser : cette forme est une sémantique précise et formelle, mais ne permet pas forcément de prédire tout le temps précisément le comportement d'un bout de code dans tous les cas. En effet, si on est en présence d'aliasing (deux noms pointant vers la

Notation (cours)	Fonction (code)	Sémantique
$a +_i b$	`add_i a b`	$\begin{cases} a+b \text{ si a et b sont des } \texttt{int} \\ \text{non-défini sinon} \end{cases}$
$a -_i b$	`sub_i a b`	$\begin{cases} a-b \text{ si a et b sont des } \texttt{int} \\ \text{non-défini sinon} \end{cases}$
$a \times_i b$	`mul_i a b`	$\begin{cases} a \times b \text{ si a et b sont des } \texttt{int} \\ \text{non-défini sinon} \end{cases}$
$a/_i b$	`div_i a b`	$\begin{cases} \text{le quotient de la division de a par b} \\ \text{si a et b sont des } \texttt{int} \\ \text{non-défini sinon} \end{cases}$
$a\%_i b$	`mod_i a b`	$\begin{cases} \text{le reste de la division de a par b} \\ \text{si a et b sont des } \texttt{int} \\ \text{non-défini sinon} \end{cases}$
$a +_f b$	`add_f a b`	$\begin{cases} a+b \text{ si a et b sont des } \texttt{float} \\ \text{non-défini sinon} \end{cases}$
$a -_f b$	`sub_f a b`	$\begin{cases} a-b \text{ si a et b sont des } \texttt{float} \\ \text{non-défini sinon} \end{cases}$
$a \times_f b$	`mul_f a b`	$\begin{cases} a \times b \text{ si a et b sont des } \texttt{float} \\ \text{non-défini sinon} \end{cases}$
$a/_f b$	`div_f a b`	$\begin{cases} a/b \text{ si a et b sont des } \texttt{float} \\ \text{non-défini sinon} \end{cases}$
$a\%_f b$	`mod_f a b`	$\begin{cases} \text{le reste de la division entière de a par b} \\ \text{si a et b sont des } \texttt{float} \\ \text{non-défini sinon} \end{cases}$

FIGURE 2.1 – Les opérations arithmétiques de base de notre machine abstraite. Dans la sémantique, les opérateurs représentent les opérations correspondantes du processeur d'un ordinateur standard (techniquement, la cible de la compilation).

même case mémoire), le comportement sera bien défini de la même manière que sans, mais le résultat sera différent. Par exemple $\rho[x \leftarrow 1][y \leftarrow 2][x \leftarrow x+y]$ n'aura pas le même effet si x et y ont la même adresse ou pas (s'ils n'ont pas la même adresse, on se retrouvera avec x valant 3 et y 2, mais s'ils ont la même adresse, on aura 4 à cette adresse).

Notation (cours)	Fonction (code)	Sémantique
$a \wedge_b b$	`and_b a b`	$\begin{cases} a \wedge b \\ \text{ si a et b sont des } \texttt{bool} \\ \text{non-défini sinon} \end{cases}$
$a \vee_b b$	`or_b a b`	$\begin{cases} a \vee b \\ \text{ si a et b sont des } \texttt{bool} \\ \text{non-défini sinon} \end{cases}$
$\neg_b a$	`not_b a`	$\begin{cases} \neg a \\ \text{ si a est un } \texttt{bool} \\ \text{non-défini sinon} \end{cases}$
$a =_m b$	`eq_m a b`	$\begin{cases} \texttt{true} \text{ si a et b sont la même valeur} \\ \texttt{false} \text{ si a et b sont différentes mais de même type} \\ \text{non-défini sinon} \end{cases}$
$a <_m b$	`lt_m a b`	$\begin{cases} \texttt{true} \text{ si a} < \text{b et a et b sont deux } \texttt{int} \\ \text{ ou deux } \texttt{float} \text{ ou si a} = \texttt{false} \text{ et b} = \texttt{true} \\ \texttt{false} \text{ si a} >= \text{b et a et b sont deux } \texttt{int} \\ \text{ ou deux } \texttt{float} \text{ ou si a et b sont deux } \texttt{bool} \\ \text{ et a} = \texttt{true} \text{ ou b} = \texttt{false} \\ \text{non-défini sinon.} \end{cases}$

FIGURE 2.2 – Les autres opérations de base de notre machine abstraite. Dans la sémantique, les opérateurs représentent les opérations correspondantes du processeur d'un ordinateur standard (techniquement, la cible de la compilation).

Ce comportement est complexe à modéliser, et on fera souvent comme s'il n'avait pas lieu, mais on ne pourra l'ignorer entièrement, puisque notre langage sera capable de créer de l'aliasing. En pratique, dans la plupart des cas, on se contentera de représenter la mémoire comme sa vue depuis un environnement et de calculer l'effet d'un bout de code en supposant qu'il n'y a pas d'aliasing. Dans ces cas, on pourra représenter l'état visible de la mémoire comme $< x \to 2, y \to 3 >$, pour signifier qu'on a un environnement où x vaut 2 et y vaut 3.

Notez qu'on peut parfaitement garder une représentation partielle de la mémoire avec plusieurs environnement tout en restant précis (à défaut sans doute d'être extrêmement lisible) : si on a deux environnements ρ et τ définissant tous deux les variables x et y, tels que x dans ρ et y dans τ pointent en réalité vers la même case mémoire, on peut le représenter ainsi :

$$< (\rho(x), \tau(y)) \to 4, \rho(y) \to 2, \tau(x) \to 7 >$$

Bon, on n'y recourra qu'en cas de besoin extrême si on n'a aucun autre moyen d'ex-

primer clairement ce que l'on souhaite. Dans la plupart des cas, on se limitera à un seul environnement à la fois.

Dans le code que nous produirons, la mémoire n'apparaîtra pas comme une valeur directement manipulable, mais sera uniquement présente via des environnement qui seront définis par le module `Util.Environment`. Ces environnement contiendront en pratique des références, et il sera possible pour plusieurs noms contenir la même référence (ou même plusieurs noms dans des environnement différents), ce qui permettra bien de modéliser l'aliasing, le tout sans avoir à modéliser des cases explicites d'une mémoire physique (tâche qu'on déléguera donc à la traduction vers l'assembleur).

Ces environnements étant utiles à plusieurs endroits du compilateurs (notamment pour les phases d'analyse sémantique), ils sont génériques et disposent de plus de fonctions que nécessaires ici. On montre les fonctions utiles pour la sémantique, ainsi que la notation du cours de celle-ci dans la figure 2.3. Ce sont essentiellement des fonctions d'accès et de modification, les cases mémoires étant manipulées séparément. Il y a également une manière de créer des alias entre deux cases mémoires de deux environnement. On notera que dans le code, cela correspond à deux fonctions du module `Util.Environment`, c'est le cas pour des raisons techniques. Cela servira pour les passage par référence des appels de fonctions (voir plus bas) [1].

On peut maintenant parler des tableaux, que nous avons laissé de coté jusqu'ici. Un tableau est encodé par la `value array`, qui contient deux éléments : un nom `name` et un environnement `env`. La signification de ceci est que le tableau stocké à cette case va avoir ses valeurs accessible via l'environnement `env`. Ses valeurs seront alors stockées dans les variables nommées `name0`, `name1`, etc. Sa taille sera stockée dans la variable nommée `namesize`. Évidemment, lorsqu'on va manipuler des tableaux, il faudra prendre soin que les noms des cases des tableaux ne se chevauchent pas avec les variables pouvant exister par ailleurs dans le programme. Pour cela, le plus simple est d'utiliser un caractère n'étant pas autorisé dans les noms de variables, ce qui est le cas du caractère '`#`'. On le précisera spécifiquement dans la partie sémantique du langage. C'est une simplification avec le comportement d'un ordinateur réel dans le sens où dans ce dernier cas, les questions d'aliasing entre deux tableaux sont parfois complexes, alors qu'ici, elle sont inexistantes par constructions (deux tableaux de noms différents ne peuvent interférer, du moins si on prend la peine de leur donner des noms avec un caractère spécial). Il peut paraître surprenant de spécifier `env` dans la valeur tableau. Dans le cadre d'un programme sans fonction, cela serait inutile, mais dans le cas d'un appel de fonction, un tableau sera passé par référence, et pour que les cases du tableau soient bien celles du tableau passé en argument, on a besoin de stocker dans la valeur tableau l'environnement dans lequel celui-ci est stocké.

En réalité ici, notre valeur `array` est réellement un pointeur : on pourrait d'ailleurs très facilement faire en sorte que la case `VArray(name,env)` pointe bel et bien vers la case `name` de `env`. Étant donné que la mémoire est indexée par des chaînes de caractères et non des entiers, il serait cependant un peu complexe de simuler le comportement exact des pointeurs en C.

Lorsqu'on manipulera des fonctions, on aura également besoin d'une mémoire stockant les déclarations de fonctions. Pour garder la sémantique la plus simple possible (et autoriser une légère surcharge), on considèrera qu'on dispose d'une seconde mémoire dédiée à stocker les fonctions qui sera une fonction de Var vers leur déclaration syntaxique. Cet environnement particulier sera noté ρ_F et sera global à un programme. On verra plus bas que c'est la seule information dont la machine a besoin pour les exécuter. Pour plus de simplicité, on gardera implicite cette mémoire quand on ne parle pas de fonctions. En réalité, au coût de

1. De même, si on manipulait des pointeurs comme en C, on en aurait besoin pour ceux-ci.

Notation (cours)	Fonction (code)	Sémantique
$\emptyset$	`new_environment ()`	Crée un environnement vide
$\rho(x)$	`get rho x`	Récupère la `value` pointée par x dans ρ.
$\rho[x \leftarrow v]$	`modify rho x v`	Place la `value` v dans la cellule pointée la variable x de l'environnement ρ.
$\rho[x \diamond \tau(y)]$	`let m = get_ref tau y in add_ref rho x m`	Lie la case pointée par x dans ρ à celle pointée par y dans τ. Cela sert à aliaser des cases mémoire entre deux environnement pour les passages par référence.

FIGURE 2.3 – Les opérations sur les environnements

définir un type plus complexe, on pourrait tout stocker dans le même environnement [2], on considèrera donc dans la suite que l'environnement est capable de stocker également des définitions de fonctions et qu'on les y récupère lorsqu'on veut appeler une fonction. Dans le code, l'environnement de fonction sera également un environnement de `Util.Environment` (mais dont les cases contiennent des déclarations de fonctions et non des `value`).

Dans un ordinateur réel, la mémoire contenant le programme et la mémoire contenant les données ne sont d'ailleurs pas différentes (du moins si on suppose que tout est en RAM ; si on commence à parler de ROM, c'est évidemment un brin plus compliqué – sans compter que les systèmes d'exploitations assurent qu'elles sont différentes pour des raisons de sécurité), et donc au final, ces fonctions sont simplement stockées quelque part dans la mémoire, et le programme associera à un nom de fonction un pointeur vers ces zones mémoires. Au final, pour un ordinateur réel, ce qui fait la différence entre une donnée et une instruction, c'est uniquement ce qu'on en fait.

Nous avons maintenant une machine abstraite entièrement définie, avec des types manipulables, des opérations sur ceux-ci, et une manière de manipuler la mémoire. Cette machine est un modèle idéalisé d'un ordinateur, mais pas si loin que ça de la réalité en terme d'expressivité des opérations (qui correspondent aux opérations de vrais processeurs). C'est surtout en terme de mémoire que notre modèle est idéalisé. Ce modèle nous permettra de définir proprement la sémantique du langage du cours. Cela signifie qu'une fois cette sémantique définie, implémenter un interpréteur ne sera qu'une implémentation directe de celle-ci, et pour le passage à un code bas-niveau, c'est essentiellement la gestion de la mémoire qui sera complexe.

2.3 Langage(s) considéré(s)

Il faut comprendre qu'un langage de programmation est un couple syntaxe/sémantique. La partie syntaxique dit comment les expressions et instructions s'écrivent. La partie sé-

2. Et d'ailleurs, si on définissait un langage fonctionnel, c'est exactement ce que l'on ferait.

mantique dit le sens de ces écritures.

Par exemple la logique propositionnelle inscrit sa syntaxe par un ensemble de règles de construction de mots. Si x et y sont deux mots de la logique propositionnelle, $(x \wedge y)$ en est aussi un. La sémantique est définie par induction pour chaque construction syntaxique. La sémantique de $(x \wedge y)$ est 1 si et seulement si la sémantique de x est 1 et la sémantique de y est également 1.

En réalité, ces notions sont communes à toutes les formes de langages, y compris les langages naturels (français, anglais, etc.) – et même ceux qui ne recourent pas à un support écrit (même si dans ce cas, la syntaxe ne parle pas de l'écrit, mais c'est un léger abus de le dire). Par exemple en français, la syntaxe va être donné par la grammaire (et l'orthographe) qui permettront de former des phrases correctes, et la sémantique correspondra à ce que veut dire la phrase (ce qui est très très grossier : les langues naturelles ont trop d'usages pour que ceci soit vraiment correct). Sur les langues naturelles, la syntaxe est déjà compliquée à définir, mais donner une sémantique précise l'est encore plus (pour ne pas dire impossible, et probablement pas souhaitable). Il est cependant possible de remarquer qu'il existe des phrases en français (grammaticalement correctes) qui n'ont pas de sens (ou du moins sont grossièrement absurde, par exemple «Le nom d'hirsute vole avec mon igname à moteur vengeur»), ce qui est une bonne analogie avec les langages de programmation où des programmes pourtant syntaxiquement corrects peuvent ne pas avoir de sémantique (par exemple `x := 4/0;`).

Dans le monde des langages de programmations, ces notions sont fort heureusement plus simples à appréhender (ils ont été conçus pour cela) : la syntaxe est suffisamment restreinte pour qu'il soit possible de déterminer automatiquement si un texte est un programme syntaxiquement correct (ce qu'on verra au chapitre 3), et surtout la sémantique a un seul sens bien précis, à savoir la définition d'une transformation d'un état de la mémoire de la machine où elle s'exécute vers un autre.

Nous allons donc définir un mini-langage en donnant séparément sa syntaxe et sa sémantique.

2.3.1 Généralités

Avant de le faire proprement dit, commençons par donner des généralités et quelques définitions qui s'appliqueront (plus ou moins) à tous les langages de programmation.

Tout d'abord, parlons de syntaxe. Un langage de programmation va être défini comme suit :

Définition 2.3.1. *Un langage de programmation L est défini par une grammaire algébrique $\mathcal{G}_L = (N, T, S, R)$.*

Un programme P de L est un arbre de dérivation de $\mathcal{G}_L$.

On ne redonne pas précisément ici la définition des grammaires algébriques (vus en MPC, et on en reparlera plus précisément au chapitre 3). Par ailleurs, en donnant notre définition de langage, on ne le donnera pas explicitement comme une grammaire, mais plutôt comme une définition inductive (ce qui est en fait la même chose).

Le point important à retenir, c'est que la définition non-ambiguë d'un programme est bien de le voir comme un arbre de dérivation d'une grammaire. C'est cela qui permet que le programme porte son sens de manière précise. La représentation sous forme de texte n'en sera qu'une représentation plus lisible par un humain (et plus facile à stocker aussi).

Les différents non-terminaux de la grammaire, correspondront à des éléments (syntaxiques, mais également avec un rôle précis) différents du langages. Ces concepts dé-

pendent évidemment du langage, mais on peut en donner deux qui reviennent dans à peu près tous les langages : les expressions et les instructions (ou statement en anglais).

Les expressions sont des éléments du langage qui auront une valeur (de type valeur de la machine abstraite) – typiquement les constantes, les variables, les expressions arithmétiques, les cases de tableaux, ou encore les appels de fonctions. Idéalement elles n'ont aucun effet sur la mémoire, mais en réalité – particulièrement dans les langages impératifs, elles en ont souvent un.

Les instructions sont des éléments du langage qui n'ont pas de valeur associée, mais qui auront un effet sur la mémoire. Typiquement ce seront les affectations, les structures de contrôle, les déclarations de variables ou de fonction, etc.

Ces deux concepts sont des catégories assez pratiques, mais qui sont assez variables d'un langage à l'autre. Par exemple, dans le langage que nous définirons, les instructions sont découpées en deux concepts de rôle différents : le déclarations de fonction d'un coté, et les instructions de l'autre. En C, on a à peu près un comportement similaire à première vue (mais beaucoup plus subtil et complexe). En OCaml, il n'y a que des expressions (qui peuvent faire des effet de bord), mais qui peuvent renvoyer une valeur d'un type spécial (`unit`) représentant l'absence de valeur.

Enfin, on peut parler de la sémantique d'un programme. À un programme, on va associer deux choses : son effet sur la mémoire et la valeur qu'il renvoie, ce qui permet de donner un cadre commun aux expressions et aux instructions. Ces deux choses dépendront évidemment de l'état de la mémoire auquel on applique le programme.

Définition 2.3.2. *Soit $\mathcal{M}$ l'ensemble des états de mémoire possibles de la machine considéré, et* `value` *l'ensemble des valeurs manipulables par la dite machine.*

La sémantique d'un programme P d'un langage L est une fonction $[\![P]\!]$ de type $\mathcal{M} \to \mathcal{M} \times$ `value`*, c'est-à-dire qui prend en argument un état de mémoire et renvoie un état de mémoire et une valeur.*

L'idée de cette définition est de considérer que le programme prend «en entrée» toute la mémoire de l'ordinateur et renvoie la valeur calculée (quand il y en a une), et la mémoire après son application. C'est évidemment une vision fonctionnelle (dans une vision impérative, on traiterait l'effet sur la mémoire par des effets de bord, ici ça permet de l'expliciter).

Elle permet de capturer tous les comportements possibles, mais dans certains langages, on va éventuellement raffiner. Par exemple, si un langage contient des expressions sans effet de bord, on pourra considérer que la sémantique d'une expression est une fonction de type $\mathcal{M} \to$ `value`. De même, si on a des instruction, on peut considérer que leur sémantique est une fonction de type $\mathcal{M} \to \mathcal{M}$. De fait, dans notre langage, tant qu'on n'aura pas ajouté de fonctions, c'est ainsi qu'on le modélisera (pour simplifier). Évidemment, pour des expressions qui font des effets de bord, il faut garder les deux.

Ces fonctions de sémantiques seront ensuite définies inductivement sur la définition du langage (qui est elle-même inductive, puisque c'est une grammaire algébrique) : les cas de base (i.e., les non-terminaux) seront associés soit à une valeur de la machine abstraite, soit à une opération de la machine abstraite. Les cas inductifs composeront les fonctions des sous-termes (par exemple, la sémantique d'un if-then-else consistera à appliquer la sémantique du then si le test est satisfait et celle du else sinon). Mais le point important à retenir est que cette sémantique est définie par rapport aux valeurs et opérations de la machine abstraite, et dépendent donc d'elle – et que lorsqu'on passera à un vrai ordinateur, cette sémantique sera au final définie par rapport aux instructions de l'ordinateur en question.

En fait, cela a des conséquences assez réelles, puisque toutes les machines n'ont pas les mêmes instructions et pas nécessairement la même représentation. Et donc, deux comportements sont possibles quand on a deux machines différente vers lesquels compiler un même langage : soit considérer que la sémantique prime, et donc le code produit peut être très différent sur les deux machines ; soit considérer qu'on veut des traductions similaires bien que le comportement soit différent. Dans ce dernier cas, la bonne manière de faire est de dire que la sémantique n'est pas définie dans ce cas. C'est exactement le choix qui est fait pour l'overflow des entiers signés en C. Si vous regardez la norme du C (qui est en gros sa sémantique, même si c'est moins formel que ce qu'on fait ici), vous verrez que la sémantique de l'overflow des entiers signés n'est pas définie (typiquement, le résultat de `INT_MAX + 1`). Vous ne l'avez probablement que peu observé car la plupart des machines modernes représentent les entiers signés avec du complément à 2, mais il existe des machines qui les représentent avec du complément à 1, et que l'overflow se comporte différemment dans ces deux cas (au niveau du processeur). Dans le but que l'addition soit toujours codée comme une seule instruction assembleur (ce qui est raisonnable, c'est pas comme si c'était une opération fréquente), le choix est fait en C de laisser le comportement de l'architecture du processeur prévaloir dans ce cas.

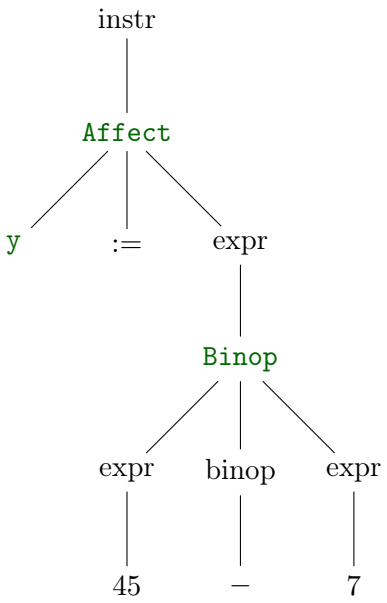
2.3.2 Syntaxe du langage du cours

Nous allons définir syntaxiquement le langage que nous chercherons à manipuler dans le cadre de ce cours. Ce langage s'exécutera sur la machine abstraite définie plus haut, la section suivante en donnera la sémantique. On pourra considérer des sous-langages de ce langage en retirant des instructions ou des expressions (on explicitera cela). Ce langage nous servira de langage intermédiaire dans le compilateur final, c'est-à-dire de langage obtenu après l'analyse sémantique et avant la traduction vers du code à 3 adresses. À ce titre, ce sera un langage avec un minimum de constructions, et qui ne vérifie pas les types des données qu'il manipule (cela sera fait par l'analyse sémantique). Il aura cependant des expressions complètes (contrairement au code à trois adresses) et différentes stratégies de passage d'argument.

Comme vu plus haut, le langage sera défini via une grammaire algébrique, et les programmes seront les arbres de dérivation de cette grammaire. On va cependant faire une simplification de plus, et considérer que la grammaire génère un arbre en éliminant les non-terminaux intermédiaires (pour plus de simplicité, étant donné que chaque «cas» sera porté par un symbole explicite).

Par simplicité et lien avec la partie implémentation, on note ces arbres comme des constructeurs de type OCaml (en fait on va noter notre langage de la même manière qu'il l'est dans le module `Course_language.Ast`).

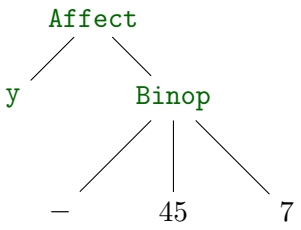
Par exemple, dans notre langage, le programme qu'on écrirait en texte par `y:= 45-7` correspondra à l'arbre de dérivation suivant :



Cet arbre n'est pratique ni à manipuler ni à afficher, aussi, on considèrera le terme obtenu en éliminant les non-terminaux, c'est-à-dire le terme généré par la grammaire, et on ne gardera pas non plus les connecteurs dont le sens est déjà porté par le constructeur (par exemple, le := qui correspond à `Affect`). Typiquement, le programme précédent sera représenté par

```
Affect(y,Binop(Sub,45,7))
```

qui ne contient que les informations nécessaires, et représenté comme un arbre donnera ceci :



À noter que pour coller au code et à une convention habituelle, dans la notation OCaml, on met l'opérateur binaire en premier argument – pour les arbres, on garde une représentation plus habituelle pour un humain.

On considèrera donc dans la suite qu'un programme est un arbre et la sémantique du programme sera définie sur cet arbre. Dans la partie implémentation, ces arbres seront représentés comme des types OCaml, et la syntaxe en sera très proche (on utilisera des types somme pour représenter les non-terminaux de la grammaire, et donc les arbres contiendront des constructeurs en plus que ceux utilisés ici, notamment pour les cas de base).

On donne une définition *inductive* de notre langage, mais c'est en réalité directement équivalent à en donner une grammaire algébrique (chaque cas correspond à une règle de réécriture) – on n'en explicite simplement pas la définition formelle (mais qui se déduit bien de ce qu'on donne ici). Cette définition est exactement celle donnée dans le module `Course_language.Ast` de l'interpréteur que nous implémenterons dans le TD lié à ce chapitre (et qui restera pour l'intégralité de ce cours).

On a vu dans la partie 2.3.1 qu'on distingue en général les expressions des instructions. C'est bien notre cas ici, mais on va également avoir d'autres types de nœuds de nos arbres (i.e., non-terminaux) qui vont représenter des éléments de notre langages qui sans être eux-même vraiment des programmes, vont en faire partie, et auront pour certains également une sémantique. Typiquement ce seront les opérateurs (qui auront une sémantique), ou les listes (qui n'en auront pas, mais seront nécessaires dans les définitions).

On utilisera des listes qui sont exactement ce que vous connaissez en programmation et sont définies comme suit :

$$A\_list \ ::= \ \texttt{[]} \mid A :: A\_list$$

pour A un non-terminal (ou terminal) quelconque, et où `[]` est la liste vide. Dans le code, on utilisera simplement le type `'a list` de OCaml, et on encourage fortement à utiliser les fonctions qui y sont définies (`map`, `fold_left`, `iter`, `iter2`, etc.).

Par convention, les non-terminaux seront notés en italique (correspondant aux types OCaml représentant les nœuds de l'arbres), et les terminaux en script de code (correspondant aux constructeurs des types et aux données).

Dans cette définition, on utilisera les types OCaml `int`, `float` et `bool` pour les constantes de base, et le type `string` pour les identifiants, ainsi que pour les chaînes de caractères à afficher (`Print_str`).

On définit les ensembles des opérateurs binaires :

$$\begin{aligned} binop \ ::= \ & \texttt{Add} \mid \texttt{Sub} \mid \texttt{Mul} \mid \texttt{Div} \mid \texttt{Mod} && \text{opérateurs arithmétiques} \\ \mid \ & \texttt{And} \mid \texttt{Or} && \text{opérateurs booléens} \\ \mid \ & \texttt{Eq} \mid \texttt{Neq} \mid \texttt{Lt} \mid \texttt{Gt} \mid \texttt{Leq} \mid \texttt{Geq} && \text{opérateurs de comparaison} \end{aligned}$$

ainsi que les opérateurs unaires :

$$unop \ ::= \ \texttt{UMin} \mid \texttt{Not}$$

On définit l'ensemble des expressions arithmétiques Arith inductivement comme suit :

$$\begin{aligned} expr \ ::= \ & \texttt{Var}(\texttt{string}) \mid \texttt{Cst\_i}(\texttt{int}) \mid \texttt{Cst\_f}(\texttt{float}) \mid \texttt{Cst\_b}(\texttt{bool}) && \text{cas de base} \\ \mid \ & \texttt{Binop}(binop, expr, expr) && \text{opérateurs binaires} \\ \mid \ & \texttt{Unop}(unop, expr) && \text{opérateurs unaires} \\ \mid \ & \texttt{Array\_val}(\texttt{string}, expr) && \text{lecture de tableau} \\ \mid \ & \texttt{Size\_tab}(\texttt{string}) && \text{taille d'un tableau} \\ \mid \ & \texttt{Func}(\texttt{string}, expr\ \texttt{list}) && \text{appel de fonction} \end{aligned}$$

On définit inductivement les instructions comme suit :

$$
\begin{array}{rll}
\text{instr} ::= & \texttt{Affect}(\texttt{string}, expr) & \text{affectation} \\
| & \texttt{Block}(instr\ \texttt{list}) & \text{séquence d'instruction} \\
| & \texttt{IfThenElse}(expr, instr, instr) & \text{if then else} \\
| & \texttt{While}(expr, instr) & \text{while} \\
| & \texttt{Affect\_array}(\texttt{string}, expr, expr) & \text{affectation de tableau} \\
| & \texttt{Array\_decl}(basic, \texttt{string}, expr) & \text{déclaration de tableau} \\
| & \texttt{Proc}(\texttt{string}, expr\ \texttt{list}) & \text{appel de procédure} \\
| & \texttt{Return} & \text{retour sans valeur} \\
| & \texttt{Return}(expr) & \text{retour d'une valeur} \\
| & \texttt{Print\_str}(\texttt{string}) & \text{affichage d'une string} \\
| & \texttt{Print\_expr}(expr) & \text{affichage d'une valeur} \\
| & \texttt{Decl}(basic, \texttt{string}) & \text{déclaration d'une variable}
\end{array}
$$

Dans ces instructions, les déclarations nécessitent de donner le type (*basic*). Ce type est défini par :

$$basic ::= \texttt{TInt} \mid \texttt{TFloat} \mid \texttt{TBool} \mid \texttt{TNull}$$

Dans le code textuel, ces types seront respectivement écrits : int, float, bool et null. Le dernier type, null, n'est pas un type de donnée, mais un type manifestant l'absence de donnée (comme le void en C ou le unit en OCaml). Il ne sera en pratique utilisée que dans les déclarations de fonctions qui ne renvoient pas de valeur.

Note : dans le code OCaml, il y a une légère subtilité : on n'a donné qu'un seul construction `Return` que peut contenir une expression ou ne pas en contenir. En OCaml, cela est matérialisé par le fait que le constructeur `Return` contient une `expr Option.t`, ce qui est exactement la même chose qu'ici (soit `None`, soit `Some expr`), mais alourdirait un peu la définition de la sémantique ici, donc on le laisse implicite (de la même manière que les constructeurs des valeurs sont laissés implicites dans ce document).

Il nous reste à définir les déclarations de fonction. On va considérer un langage pouvant passer des valeur par *adresse* et par *référence*, nous avons donc besoin de machinerie.

On définit un paramètre comme un triplet contenant un type de paramètre, un type et un nom :

$$param ::= (type\_param, type, \texttt{string})$$

où *type* est défini par :

$$\texttt{Basic}(basic) \mid \texttt{Array}(basic)$$

et *type_param* par

$$type\_param ::= \texttt{Reference} \mid \texttt{Value}$$

Dans le texte, un type `Basic(b)` sera simplement noté `b`, et un type `Array(b)` sera noté `b[]`. Le type de paramètre sera noté par la présence du mot-clé `var` pour les paramètres par références et rien pour ceux par valeur.

Une déclaration de fonction est définie par :

$$declFunc ::= \texttt{DeclF}(basic, \texttt{string}, param\ \texttt{list}, instr)$$

Un programme au sens le plus général est une liste de déclarations de fonctions, mais dans les restrictions sans appels de fonctions, on considèrera un programme comme étant une instruction.

On considère dans la suite les restrictions suivantes :

- L_{calc} **:** Le langage avec uniquement les expressions sans appels de fonction ni tableau, et uniquement des affectations et des blocs.
- L_{branch} **:** L'ajout des `if then else` et des `while`.
- L_{array} **:** L'ajout des tableaux.
- $L_{function}$ **:** L'ajout des fonctions.

On peut évidemment considérer chaque extension indépendamment. Néanmoins, dans la suite de ce chapitre, on les considèrera dans cet ordre. Lors de l'implémentation, on suggère également de suivre cet ordre.

2.3.3 Sémantique

On définit maintenant la sémantique de ce langage. On va procéder en commençant par la restriction L_{calc} en expliquant pour chaque ajout ce qu'il faut ajouter.

Dans le corps du texte, on va essentiellement commenter les concepts et les difficultés. Étant donné que donner une sémantique formelle est un peu rébarbatif et long en français, on a regroupé les syntaxes et sémantiques des différents éléments du langage dans les figures 2.4, 2.5, 2.6, 2.7.

Vous noterez que ces sémantiques sont définies avec les éléments de la machine abstraite définie plus haut dans ce chapitre.

Expressions simples et affectations (L_{calc})

On définit d'abord inductivement la valeur d'une expression arithmétique *expr* dans une environnement ρ : $[e]_\rho$, qu'on considère dans un premier temps comme étant de type $\mathcal{M} \to$ `value`. En effet, nos expressions ne font pas d'effet de bord, à l'exception des appels de fonctions, qu'on n'ajoutera que plus tard.

De même, on va dans cette sémantique uniquement représenter les états de mémoire par un environnement ρ. Ainsi, si techniquement il faudrait représenter un état de mémoire par l'état complet et l'environnement (qui attribue chaque nom à une case), on simplifiera en ne mettant que l'environnement. Et là aussi, quand on ajoutera les appels de fonctions, il faudra préciser ça.

Au final, en toute rigueur, on devrait définir la sémantique d'une expression de la forme suivante : $[\![e]\!](M, \rho) = ((M', \rho'), v)$. Mais comme ici M et ρ seront toujours égales à M' et ρ' et qu'on représentera le couple (M, ρ) par ρ uniquement, on écrira au final $[e]_\rho = v$ à la place.

Avant cela, il faut donner une sémantique aux opérateurs qui vont naturellement s'envoyer sur des opérations de la machine abstraite. Dans certains cas (notamment les opérateurs unaires et les opérateurs de comparaison), la sémantique ne sera pas directement une opération, mais s'exprimera avec plusieurs d'entre elles.

La sémantique des opérateurs binaires est une fonctions prenant deux `value` et renvoyant une `value`. Il n'y a pas de difficulté particulière sur ces sémantiques, on renvoie donc à la figure 2.4 pour le détail. Notez simplement que les opérateurs arithmétiques, ont deux cas (`int` et `float`) à considérer (donc le code contiendra un test), et que la machine abstraite contenant moins d'opérateurs de comparaison que le langage, la plupart d'entre eux sont composés à partir de ceux de la machine.

Les deux opérateurs unaires ne présentent pas non plus de difficulté particulière. Leur sémantique est une fonction de type `value -> value` et est présentée en figure 2.5.

Étant donné cette sémantique pour les opérateurs, la valeur d'une expression de L_{calc} se définit sans difficultés : la valeur d'une constante est la `value` correspondante, et les

opérateurs binaires et unaires sont l'application de la sémantique de l'opérateur à la valeur des (de l') expressions présentent dans l'opérateur. Cela est résumé en figure 2.6.

Insistons sur le fait que cette définition est importante et que le `Add` syntaxique (qui n'est qu'une notation) et sa sémantique (qui est une fonction) n'ont pas le même statut. De la même manière, et moins intuitivement, l'expression `Cst_i(4)` n'est pas la même chose que sa sémantique qui est une valeur de la machine abstraite. C'est sans doute assez peu intuitif, mais je pense pouvoir illustrer cela en nous ramenant à un exemple en français : le mot «sandwich» n'est pas un sandwich et ne vous permet pas de vous rassasier : c'est juste un mot. Alors que sa sémantique (un sandwich donc) est bel est bien quelque chose que vous pouvez manger. La distinction entre l'expression `Cst_i(4)` et sa sémantique est de même nature (sauf que la valeur 4 ne se mange pas [3], mais j'espère que vous saisissez l'idée).

De même, remarquons que ce n'est pas la seule sémantique possible de ces expressions (on pourrait donner leur sémantique sur des machines abstraites travaillant sur $\mathbb{Z}$ ou encore sur les entiers 16 bits, auquel cas on aurait des opérations différentes), même en gardant des sémantiques «naturelles» (on pourrait très bien définir la sémantique de $e_1 + e_2$ comme étant $\max(e_1 - e_2, e_2 - e_1)$, mais ce ne serait pas très raisonnable pour un langage de programmation). Cela permet par contre de définir des langages avec plus d'opérateurs de base que la machine sur laquelle ils s'exécutent.

Usuellement, on aura des opérations arithmétiques qui correspondent à ce que savent faire les processeurs pour avoir une correspondance naturelle entre nos langages et les langages machines et qui corresponde à notre intuition [4].

On peut maintenant définir la sémantique d'une instruction d'un programme (même si pour l'instant nous n'en avons qu'une). La sémantique d'une instruction est une fonction des configurations dans les configurations, c'est-à-dire une action qui modifie la mémoire de la machine abstraite, donc une fonction de type `value Util.Environment -> value Util.Environment`. Cette sémantique est résumée en figure 2.7. De même, c'est une simplification par rapport à la notion de sémantique générale due au fait qu'une instruction ne renvoie jamais de valeur. Et tout comme pour les expressions, on a simplifié la mémoire pour ne la représenter que via un environnement. Lorsqu'on fera les appels de fonctions, si on veut être précis, il faudra faire réapparaître explicitement la séparation entre la mémoire et l'environnement.

Dans l'implémentation de cette sémantique, au lieu de modéliser ces sémantiques comme des fonctions de type `value Util.Environment -> value Util.Environment`, on les modélisera plutôt en faisant des effet de bord sur l'argument de la fonction (et donc la sémantique sera de type `value Util.Environment -> unit` (enfin, en réalité il y a d'autres arguments, mais je simplifie)). Cela permet d'éviter de copier les environnements (ce qui serait coûteux), mais également de pas mal simplifier ce qu'il se passera lorsqu'on ajoutera des fonctions : on le verra d'ailleurs à la section correspondante que si l'approche choisie ici est plus simple à comprendre que des effets de bords sur les constructions de base, ce n'est pas tellement le cas quand on a des fonctions. Cela permettra également d'avoir des effets de bords sur les expressions (via les appels de fonctions).

La sémantique de l'affectation correspond assez naturellement à une substitution, puisqu'elle calcule la valeur de e, puis la stocke dans la case mémoire pointée par `x` :

$$[\![\texttt{Affect}(\texttt{x}, e)]\!](\rho) := \rho[\texttt{x} \leftarrow e]$$

3. Sauf bien entendu si elle est suivie de «quarts».

4. Je vous laisse juger de l'utilité d'un langage de programmation où la notation «x+y» dénoterait le carré de x auquel on soustrait $y^x - 4$.

La sémantique d'un bloc se définit inductivement en appliquant successivement les instructions qu'il contient comme suit :

$$\begin{aligned}[\![\texttt{Block([])}]\!](\rho) &:= \rho \\ [\![\texttt{Block(i::t)}]\!](\rho) &:= [\![\texttt{Block(t)}]\!]([\![\texttt{i}]\!](\rho))\end{aligned}$$

En OCaml, on pourra utiliser `fold_left` pour définir simplement la sémantique d'une liste : `List.fold_left sem config p`, où `sem` est une fonction implémentant la sémantique d'une instruction, `config`, une configuration, et `p` une liste d'instructions.

Parlons tout de suite des fonctions d'affichage : celles-ci sont là pour pouvoir avoir un affichage dans notre interpréteur (qui serait sinon bien triste). Insistons sur le fait que malgré qu'on puisse afficher des string, notre machine ne peut les manipuler comme un type de donnée, ce ne sont que des chaînes constantes et une commande particulière. Ainsi, la sémantique de `Print_str` ne modifie pas l'environnement et donc $[\![\texttt{Print\_str}]\!](\rho) = \rho$.

Pour `Print_expr`, on a le même phénomène, à part qu'il faut évaluer l'expression pour l'afficher, et que, s'il y a des appels de fonctions, cela peut faire des effets de bord (si on avait des appels de fonction). Comme dit plus tôt, le formalisme choisi ici ne permet pas de faire cela simplement, mais on peut rendre cela clair (quoiqu'informel) en écrivant : $[\![\texttt{Print\_exp}(expr)]\!](\rho) = \rho$ après avoir appelé $[expr]_\rho$.

Dans l'implémentation de l'interpréteur, il faudra évidemment appeler des fonctions d'affichage, mais c'est un effet de bord qu'on ne modélise pas ici, pour simplifier.

Si on voulait l'écrire en tenant en compte tous les effets de bords possibles et expliciter l'affichage, il faudrait écrire (en supposant que la machine abstraite dispose d'une instruction print ne modifiant pas la mémoire) :

$$[\![\texttt{Print\_exp}(expr)]\!](M, \rho) := \text{soit } ((M', \rho'), v) = [\![expr]\!](M, \rho); \texttt{print}(v); (M', \rho')$$

Tests et boucles (L_{branch})

Pour les ajouts apportés par L_{branch}, il suffit d'ajouter la sémantique du `if then else` et du `while`. Le `if then else` est assez simple : si le test est évalué à vrai, alors la branche du `then` est prise, si le test est évalué à faux, la branche du `else` est prise, et si le test n'est pas un booléen, la sémantique n'est pas définie. Ce dernier point signifie qu'on laisse la liberté à l'implémentation de faire ce qu'elle veut. En pratique, la machine abstraite stoppera le programme (via une exception), mais dans une traduction en langage machine, il serait plus compliqué de faire cela, et on pourrait avoir une erreur silencieuse et un programme qui renvoie un résultat absurde (vous avez probablement déjà observé cela en C). En pratique, sur un `if then else`, une des deux branches sera exécutée (très probablement le `then`, puisque seul l'octet 0 représente faux en assembleur).

$$[\![\texttt{IfThenElse}(expr, instr1, instr2)]\!](\rho) = \begin{cases} [\![\texttt{instr1}]\!](\rho) \text{ si } [expr]_\rho = \texttt{true} \\ [\![\texttt{instr2}]\!](\rho) \text{ si } [expr]_\rho = \texttt{false} \\ \text{non-défini, sinon} \end{cases}$$

Pour la sémantique du `while`, c'est une peu plus compliqué : la définition de la sémantique est récursive. En effet, le nombre d'itération du `while` n'est pas connu a priori, et le programme peut très bien ne pas terminer (par exemple `While(true,Block([])` ne terminera pas). Il y a plusieurs manière de définir proprement cette sémantique, on adopte ici une sémantique dite «à petit pas» puisqu'elle exprime la sémantique du `while` en fonction d'un seul tour de boucle en se rappelant récursivement :

$$[\![\texttt{While}(expr, instr)]\!](\rho) = \begin{cases} [\![\texttt{While}(expr, instr)]\!]([\![instr]\!](\rho)) \text{ si } [expr]_\rho = \texttt{true} \\ \rho \text{ si } [expr]_\rho = \texttt{false} \\ \text{non-défini sinon} \end{cases}$$

La sémantique n'est définie que si le calcul ci-dessus termine (une boucle infinie n'a pas de sémantique). Remarquez que c'est le seul cas où la sémantique est définie en fonction de celle d'une instruction qui n'est pas une sous-instruction de celle dont on définit la sémantique. Ce n'est pas un hasard puisque c'est la seule qui peut ne pas terminer [5]. L'avantage de cette sémantique est qu'elle correspond très bien à ce qui se passera dans le code assembleur qu'on génèrera dans quelques chapitre, puisque les saut de retour de boucle correspondront bien à exécuter le programme décrit ci-dessus. Elle est donc plus proche de l'implémentation, ce qui nous simplifie normalement sa compréhension.

On peut définir une sémantique «à grand pas» qui permettrait d'avoir une seule expression, mais elle repose sur la notion de *plus petit point fixe* dans un treillis, sur laquelle on ne s'étendra pas plus avant, mais remarquez tout de même que ce n'est pas étonnant : un point fixe d'une fonction f, c'est une valeur v telle que $f(v) = v$, et les environnements qui sont obtenus comme résultats de la sémantique de while sont nécessairement des points fixes, puisque la seconde règle ne modifie pas la sémantique en entrée et que c'est la seule manière de terminer. Si on rappelait la même boucle sur cet environnement, il ne serait pas modifié, et c'est donc bien un point fixe.

Tableaux (L_{array})

Pour définir la sémantique des tableaux, il va falloir jongler avec plusieurs environnements (puisque les tableaux sont représentés par des pointeurs en mémoire, qui ne pointent pas nécessairement vers le même environnement). Il faudra donc, dans le cas de l'affectation, réexpliciter la distinction entre mémoire et environnement qu'on a laissée implicite jusqu'ici.

Par ailleurs, dans cette définition, on n'a pas de problème d'allocation (la mémoire étant supposée infinie, et les tableaux ne pouvant par construction se chevaucher s'ils ont des points différents). Lors d'une implémentation vers de l'assembleur, ce serait un point à traiter particulièrement soigneusement (en utilisant les routines du système d'exploitation).

Ici, pour que ce soit clair, on rappelle les cas où la sémantique n'est pas définie, mais en réalité, c'est dans la machine abstraite que les cas d'erreurs sont traités (et donc dans l'implémentation de l'interpréteur, il n'y aura pas à les considérer). Dans le rappel de la sémantique (figures 2.6 et 2.7), ces cas n'apparaissent pas.

On va forcer (dans la sémantique) que les tableaux soient stockés dans des noms qui sont nécessairement disjoints des noms de variables du programme (pour éviter les chevauchements non voulus). La solution choisie est que les noms des cases mémoire représentant le tableau contiennent le caractère '#' (dont on promet qu'il ne peut pas apparaître dans les noms de variable). C'est la déclaration des tableaux qui forcera ce nom, aussi il n'apparaîtra que là.

Commençons par la valeur de l'expression $\texttt{Array\_val}(x, expr)$:

$$[\texttt{Array\_val}(x, expr)]_\rho = \begin{cases} \tau(\mathrm{N}v)\text{si } \rho(x) = \texttt{array}(\mathrm{N}, \tau) \text{ et } [expr]_\rho = v \\ \text{et que } v \text{ est un } \texttt{int} \\ \\ \text{non-défini sinon} \end{cases}$$

5. Tant qu'on n'ajoute pas des fonctions récursives qui peuvent aussi ne pas terminer.

Simplement, si la cellule x contient un tableau et que l'expression est un entier, on renvoie la valeur stockée dans la case correspondante (i.e., l'adresse Nv de l'environnement où est le tableau), et dans le cas contraire, la sémantique n'est pas définie (puisque soit on ne regarde pas dans un tableau, soit on regarde un numéro de case qui n'est pas un entier, ce qui n'a pas de sens).

Par exemple, si on a l'expression `Array_val`(x,52) et que $\rho(x) = \texttt{array}(\text{"toto\#"}, \tau)$, alors la valeur demandée est stockée dans l'environnement τ, à l'adresse "toto#52" (attention, il faudra évidemment convertir l'entier en chaîne de caractères et le concaténer derrière le nom).

À noter que si on regarde une case en-dehors de la plage allouée du tableau, on renvoie la valeur à la case correspondante (qui n'est probablement pas allouée, donc on aura des surprises), ce qui est le même comportement qu'en C.

Un autre point à noter, qui n'apparaît pas dans la définition plus haut (bien que ce soit normalement naturel) : l'expression `expr` est évaluer *avant* de regarder la valeur `Nv` dans τ. Dans le cas où l'évaluation de `expr` produit des effets de bords, il conviendrait de spécifier que l'évaluation de $\tau(\texttt{Nv})$ se fait sur la mémoire après cette évaluation.

La taille d'un tableau est simplement stockée dans une case mémoire particulière :

$$[\texttt{Size\_tab}(x)]_\rho = \begin{cases} (\tau(\text{Nsize})) \text{ si } \rho(x) = \texttt{array}(\text{N}, \tau) \\ \text{non-défini sinon} \end{cases}$$

Pour l'affectation d'une valeur d'un tableau, on a une sémantique très similaire à l'affectation, cependant, on commence à avoir un cas où la limitation de notre choix de laisser la mémoire implicite en ne manipulant que les environnements limite l'expression correcte de la sémantique dans le cas où le tableau ne fait pas référence au même environnement que celui où on applique l'instruction. Pour cette instruction, on va donc donner une sémantique où la mémoire et l'environnement apparaissent. Tout comme pour la lecture, on ne vérifie absolument pas ici que la case où on écrit est bien entre 0 et la taille du tableau.

$$[\![\texttt{Affect\_array}(x, expr1, expr2)]\!](M, \rho) = \begin{cases} (M[\tau(\text{N}v) \leftarrow [expr2]_\rho], \rho), \\ \text{ si } \rho(x) = \texttt{array}(\text{N}, \tau) \text{ et } [expr1]_\rho = v \\ \\ \text{non-défini sinon} \end{cases}$$

Enfin, il reste à définir l'allocation d'un tableau, qui consiste simplement à fixer la variable de taille du tableau.

$$[\![\texttt{Array\_decl}(basic, x, expr)]\!](\rho) = \begin{cases} \rho[x \leftarrow \texttt{array}(\text{x\#}, \rho)][\text{x\#size} \leftarrow v] \\ \text{ si } [expr]_\rho = v \text{ qui est un } \texttt{int} \\ \text{non-défini sinon} \end{cases}$$

Notez que c'est bel et bien le seul endroit où le # apparaît explicitement, et que ici deux cases mémoires sont modifiée : celle qui contient la déclaration du tableau (x) et celle qui contient sa taille (x#size). De même, on déclare le tableau dans l'environnement courant.

Ici, il est à noter que le type de données du tableau importe peu pour l'allocation puisque la machine abstraite est capable de manipuler des types `value` de manière transparente. De même, comme notre machine abstraite dispose d'une mémoire simplifiée, on n'est pas capable d'exprimer l'allocation des cases du tableau (qui sont ici laissées non initialisées). Évidemment quand on compilera vers de l'assembleur, il sera important de tenir compte

de ce type puisqu'ils n'occupent pas tous la même taille en mémoire, et on devra donc rajouter cette information dans le code produit, de même qu'il faudra faire appel aux fonctions d'allocations du système d'exploitation.

De même, on aura, pour les déclarations de variables :

$$[\![\texttt{Decl}(basic, x)]\!](\rho) = \rho$$

puisqu'on considère que les allocations se font implicitement, et que chaque valeur occupe la même place en mémoire. Lors du chapitre de génération de code, il faudra par contre en tenir compte. En réalité, l'intérêt principal de la déclaration est surtout de permettre la vérification des types des variables lors de la phase qu'on explorera dans le chapitre 4.

Fonctions ($L_{function}$)

Les fonctions sont sans doute la partie de notre langage pour laquelle il est le plus technique de définir une sémantique, puisqu'on va devoir préparer convenablement les environnements d'appels des fonctions. De plus, les fonctions peuvent faire des effets de bord sur l'environnement d'appel de la fonction, ce qui va nous forcer à reprendre la définition complète de la sémantique (qui considère séparément la mémoire et les environnements).

Commençons par donner la sémantique d'une définition de fonction : elle consiste simplement en ajouter la définition syntaxique de la fonction à la liste des fonctions disponible. On rappelle que cette mémoire s'appelle ρ_{F}. On a donc

$$[\![\texttt{DeclF}(typ, f, params, instr)]\!](\rho_F) = \rho_F[f \leftarrow (typ, params, instr)]$$

Les déclarations de fonctions ne se font pas au même niveau que l'exécution de l'interpréteur : elles seront évaluées avant d'évaluer le point d'entrée du programme. Comme les définitions de fonctions ne peuvent pas apparaître à l'intérieur d'une fonction, on peut donc considérer que lorsqu'on exécute une instruction, ρ_F est fixée et ne peut être modifiée. Ce ne serait évidemment pas le cas dans un langage fonctionnel tel que Ocaml (ou même impératif où on autoriserait des déclarations de fonctions comme instruction), où l'environnement des définitions de fonctions serait modifié de la même manière que l'environnement qu'on a manipulé jusqu'ici. Mais ici, on peut considérer que ρ_F existe quand on exécute les appels de fonctions. Il faudra évidemment dans le code de l'interpréteur avoir cet environnement comme argument.

On définit maintenant la sémantique d'un appel de fonction $\texttt{Func}(f, args)$. Elle sera évidemment définie en fonction de $\rho_F(f)$. On suppose dans la discussion que $\rho_F(f) = typ, params, instr$. Dans le cas où $\rho_F(f)$ est vide, la sémantique d'un appel de la fonction x ne sera évidemment pas définie.

On définit d'abord un environnement d'appel, ainsi que l'effet sur la mémoire, à partir de $params$, $args$, de l'environnement de l'appelant ρ et de la mémoire initiale M et de l'environnement en construction η. Pour les arguments passés par valeur, on place simplement la valeur de l'argument comme valeur de la variable le représentant. Pour les arguments passés par référence, on utilise l'opération d'alias de la machine (cf figure 2.3 – dans notre code, cela consistera simplement à affecter la même référence aux deux variables) pour lier l'argument à la variable le représentant.

L'environnement d'appel sera défini par une fonction $\tau(params, args, \rho, M, \eta)$, où $params$ est la liste de paramètres de la fonction appelée, $args$ la liste d'arguments fournis, ρ l'environnement d'appel, M la mémoire de l'ordinateur, et η un accumulateur représentant

l'environnement que l'on construit. Cette fonction est définie récursivement (sur chaque couple paramètre, argument) :

$$\begin{aligned}
\tau(\texttt{[]}, \texttt{[]}, \rho, M, \eta) &= M, \eta \\
\tau((\texttt{Reference}, t, x) :: ld, \texttt{y} :: l, \rho, M, \eta) &= \tau(ld, l, \rho, M, \eta[x \diamond (\rho, y)]) \text{ si y est une variable} \\
\tau((\texttt{Value}, t, x) :: ld, \texttt{e} :: l, \rho, M, \eta) &= \tau(ld, l, \rho, M[\eta(x) \leftarrow [e]_\rho], \eta) \text{ en supposant que} \\
&\quad \eta(x) \text{ est une case mémoire fraîche.}
\end{aligned}$$

Si *args* et *params* ne sont pas de même taille, cet environnement n'a pas de définition (et l'appel n'a pas de sémantique définie i.e., le programme plante).

Attention, pour des raisons de simplicité lors du passage à l'assembleur, on fixe l'ordre d'évaluation des arguments de *droite à gauche*. Cela implique qu'il faudra appeler la fonction précédente sur le *miroir* des deux listes *params* et *args*. Lorsqu'on implémentera ce nouvel environnement cependant, il conviendra de bien faire attention à respecter cet ordre, et pour cela, il sera sans doute nécessaire de parcourir les listes de droite à gauche (soit avec un `fold_right`, soit en les inversant avec `rev`).

On ne peut pas passer par référence une expression qui n'est pas une variable (i.e., pas une case mémoire), et la sémantique n'est pas définie dans ce cas. Il serait en réalité possible de définir un langage où on pourrait passer par référence des cases de tableau. Pour la définition théorique, ainsi que l'implémentation de notre interpréteur cela ne changerait pas grand chose (et ce sera l'une des extensions qui vous sera proposée en TD), néanmoins, pour le passage vers un assembleur qui stocke les tableaux dans une partie différente de la mémoire que les variables (et même pour notre code trois adresses), il serait bien plus complexe de garder le même comportement. Aussi, notre langage interdit de passer des cases de tableau par référence.

À noter une autre conséquence de notre sémantique quand on parle de tableaux : si on passe un tableau par valeur et qu'on modifie l'une de ses cases, l'effet sera visible par la fonction appelante. Si l'on regarde en détail la sémantique, cela n'est pas une surprise : en effet, le tableau est représenté par son adresse dans un environnement qui est visible par la fonction appelante (en réalité, c'est un pointeur), et si le tableau a pour valeur $\texttt{array}(\texttt{name}, \rho)$, alors, si on en modifie la case 0 dans une fonction où il est passé comme paramètre, on modifie bien la case `name0` de l'environnement ρ, et non de celui de la fonction. Comme le tableau a exactement la même adresse dans la fonction appelante, elle verra bien la modification.

Ce comportement ne devrait pas vous surprendre : dans les langages que vous connaissez (C, java, et même OCaml (si on utilise des tableaux)), c'est exactement ce qui se passe. La raison en est que c'est le pointeur qu'on passe par valeur, pas la zone pointée.

Si on voulait passer le contenu d'un tableau par valeur, il faudrait réaliser une copie de tout le tableau, et cette opération est coûteuse (linéaire en la taille du tableau). Il est en général peu souhaitable que des instructions de base d'un langage ait un coût caché important, d'où le choix de laisser la responsabilité au programmeur de faire ce choix (et également, car cela en simplifie l'implémentation).

Pour les appels de fonctions, on va utiliser une variable spéciale (qu'on interdira comme variable possible dans le langage), `#result`, pour représenter la valeur de retour. Il suffit simplement de regarder la valeur de `#result` après l'appel de la fonction sur un environnement frais calculé avec τ. Cette valeur sera affectée par l'instruction `Return`.

Enfin, comme annoncé plus tôt, puisque nos fonctions peuvent faire des effets de bords et manipulent plusieurs environnement, il faut reprendre la définition générale de la sémantique pour définir le tout clairement.

La sémantique de l'appel est

$$[\![\texttt{Func}(f, args)]\!](M, \rho) = \begin{cases} ((M_3, \rho), \rho_3(\texttt{\#result})) \text{ avec } (M_3, \rho_3) = [\![\texttt{body}]\!](M_2, \rho_2), \\ \text{où } (M_2, \rho_2) = \tau(\texttt{rev}(params), \texttt{rev}(args), \rho, M, \emptyset) \\ \text{si } \rho_F(x) = typ, params, \texttt{body} \text{ et } args \text{ et } params \text{ ont la même taille} \\ \\ \text{non-défini sinon} \end{cases}$$

La sémantique d'un appel de procédure est évidemment identique, à part qu'on n'utilise pas la valeur de retour :

$$[\![\texttt{Proc}(f, args)]\!](M, \rho) = \begin{cases} (M_3, \rho) \text{ avec } (M_3, \rho_3) = [\![\texttt{body}]\!](M_2, \rho_2), \\ \text{où } (M_2, \rho_2) = \tau(\texttt{rev}(params), \texttt{rev}(args), \rho, M, \emptyset) \\ \text{si } \rho_F(x) = typ, params, \texttt{body} \text{ et } args \text{ et } params \text{ ont la même taille} \\ \\ \text{non-défini sinon} \end{cases}$$

Les instructions `Return` et `Return_v`($expr$) ont une sémantique particulière, dans le sens où elles terminent l'exécution de la fonction. Pour leur effet immédiat, c'est assez simple : $[\![\texttt{Return\_v}(expr)]\!](\rho) = \rho[\#result \leftarrow [expr]_\rho]$, et $[\![\texttt{return}]\!](\rho) = \rho$, mais de plus, elles terminent l'exécution de la fonction.

Si on veut le définir formellement, il faut alors modifier toute notre notion de sémantique de manière à ce que les objets manipulés soient non plus seulement un environnement, mais un couple constitué d'une sémantique et d'un élément pouvant être soit `Normal`, soit `Stop`. Lorsque cet élément est `Normal`, la sémantique est définie comme dans ce qui précède (en conservant `Normal`), lorsqu'il est `Stop`, toutes les sémantiques précédentes n'ont aucun effet. Les instructions `Return` et `Return`($expr$) placent alors cet élément sur `Stop`, ce qui permet donc qu'elles soient la dernière instruction à avoir un effet. Les instructions d'appel de fonction et de procédure ne sont pas impactés par cette modifications.

Lorsque l'on implémentera cela dans l'interpréteur, on utilisera des exceptions qui se trouvent avoir exactement la même sémantique, sans avoir à préciser les deux cas dans toutes les fonctions qui implémentent la sémantique des instructions. Mais tout cela est un détail d'implémentation dans les langage haut-niveau qu'on utilise pour implémenter l'interpréteur. En réalité, en assembleur c'est un comportement tout à fait naturel, qui correspond à une instruction assembleur (`ret` en x86) qui consiste simplement à sauter à l'adresse de retour (et à modifier le registre de pile).

La sémantique d'un programme, c'est-à-dire une liste de déclaration de fonctions est finalement la sémantique de la fonction nommée *main* sur un environnement de départ avec les fonctions de la liste comme fonctions appelables. En fonction de ce qu'on souhaite autoriser, on peut considérer deux sémantiques un peu différentes : soit `main` n'a pas d'arguments, auquel cas la sémantique est $[\![\texttt{body\_main}]\!](\emptyset)$, ou cette fonction a le droit d'avoir des arguments, auquel cas, on traitera ce cas comme un appel de procédure avec des arguments fixés par le programme appelant.

Au final, plus formellement, si on appelle $args$ la liste d'arguments fournis à l'interpréteur, la sémantique d'un programme prg qui est une liste de déclaration de fonctions dont la fonction *main* a pour paramètres $params$ et pour corps $body$ est :

$$[\![prg]\!] = [\![body]\!](\tau(params, args, \rho, M, \emptyset))$$

avec les définitions de fonctions ρ_F qui ont été obtenues en exécutant la liste des fonctions de prg, ρ est un environnement quelconque fourni par l'interpréteur (qui permet d'observer des effets de bord), et M la mémoire de l'ordinateur au moment de l'appel (qui dans l'interpréteur sera vide).

Dans notre interpréteur, la sémantique du programme vous sera fournie (pour traiter les arguments ce qui demande une analyse textuelle qui n'est pas le sujet de ce chapitre).

Un programme ne contenant pas de fonction *main* ne fait rien.

2.4 Compléments

2.4.1 Stratégies d'évaluation des paramètres

Dans le langage défini dans ce cours, on a vu deux stratégies d'évaluation des paramètres des appels de fonction, par valeur et par référence. Dans cette section, on va commenter un peu ces choix par rapport à ce que vous connaissez, et en parler de quelques autres.

Stratégies strictes (ordre applicatif, appel par valeur)

Le passage par valeur et le passage par référence sont des stratégies d'évaluation strictes, qui correspondent à évaluer les paramètres d'une fonction avant d'éxécuter la dite fonction.

C'est le type de stratégie assez naturel quand on considère un langage impératif et le fonctionnement intuitif d'un ordinateur, mais ce n'est pas le seul possible (voir plus loin).

L'avantage est qu'il est relativement peu coûteux à effectuer sur une machine standard (il suffit d'une pile pour stocker les valeurs des paramètres), mais son défaut est essentiellement que si l'évaluation d'un paramètre provoque une erreur, alors l'appel de la fonction en provoquera un, quand bien même la fonction n'aurait pas utilisé ce paramètre.

À noter que dans ce type de stratégie, en présence d'effets de bords, l'ordre d'évaluation des paramètre (de gauche à droite ou de droite à gauche) a une importance. Les deux ordres existent, que ce soit spécifié ou non (par exemple, java évalue les paramètres de gauche à droite, et c'est spécifié dans la norme java ; alors qu'OCaml ou C ne spécifie pas l'ordre d'évaluation – en pratique, OCaml les évalue de droite à gauche).

À partir de là, il y a des distinctions à faire dans ce qu'on passe exactement en paramètre d'une fonction.

Passage par valeur Comme son nom l'indique, le passage par valeur crée une copie indépendente du paramètre et fournit cette copie à la fonction (typiquement en la plaçant sur la pile). La conséquence de cela est qu'aucun effet de bord effectué sur cette valeur ne sera visible par la fonction appelante. L'autre conséquence est que si on fait du passage par valeur pure sur un langage acceptant de définir des types de données complexe, l'appel de fonction peut conduire à copier de grosses plages mémoire (c'est typiquement le cas en C).

Passage par référence En contraste du précédent, au lieu de placer une copie sur la pile, on place une adresse mémoire que la fonction ira modifier directement. C'est ce qui se passe dans notre langage. Cette stratégie est assez ancienne, et est assez proche de ce qu'on peut faire en assembleur – puisqu'en réalité, la mémoire reste accessible à tout moment. Cette technique ne réalisant aucune copie elle assure que l'appel d'une fonction garde un coût faible. Néanmoins, cela empêche d'avoir une mémoire bien segmentée par défaut et de bien garder trace des effets de bords pouvant survenir, et est donc source de bugs. Par

Texte	OCaml	Sémantique : $[\![op]\!](a, b)$
+	`Add`	$\begin{cases} a +_i b \text{ si a et b sont des int} \\ a +_f b \text{ si a et b sont des float} \\ \text{non-défini sinon} \end{cases}$
-	`Sub`	$\begin{cases} a -_i b \text{ si a et b sont des int} \\ a -_f b \text{ si a et b sont des float} \\ \text{non-défini sinon} \end{cases}$
*	`Mul`	$\begin{cases} a \times_i b \text{ si a et b sont des int} \\ a \times_f b \text{ si a et b sont des float} \\ \text{non-défini sinon} \end{cases}$
/	`Div`	$\begin{cases} a/_i b \text{ si a et b sont des int} \\ a/_f b \text{ si a et b sont des float} \\ \text{non-défini sinon} \end{cases}$
%	`Mod`	$\begin{cases} a\%_i b \text{ si a et b sont des int} \\ a\%_f b \text{ si a et b sont des float} \\ \text{non-défini sinon} \end{cases}$
&&	`And`	$a \wedge_b b$
||	`Or`	$a \vee_b b$
=	`Eq`	$a =_m b$
<>	`Neq`	$\neg_b(a =_m b)$
<	`Lt`	$a <_m b$
>	`Gt`	$b <_m a$
<=	`Leq`	$(a =_m b) \vee_b (a <_m b)$
>=	`Geq`	$(a =_m b) \vee_b (b <_m b)$

FIGURE 2.4 – Syntaxe et sémantique de *binop*

Texte	OCaml	Sémantique : $[\![op]\!](a)$
-	`UMin`	$\begin{cases} 0 -_i a \text{ si a est un int} \\ 0.0 -_f a \text{ si a est un float} \\ \text{non-défini sinon} \end{cases}$
!	`Not`	$\neg_b a$

FIGURE 2.5 – Syntaxe et sémantique de *unop*

Texte	OCaml	Valeur : $[e]_\rho$
i (un entier)	`Cst_i(i)`	l'int i (`VInt i`)
f (un flottant)	`Cst_f(f)`	le float f (`VFloat f`)
b (true ou false)	`Cst_b(b)`	le bool b (`VBool b`)
x (un identifiant)	`Var(x)`	$\rho(x)$
e1 op e2	`Binop(op,e1,e2)`	$[\![op]\!]([e1]_\rho, [e2]_\rho)$
op e	`Unop(op,e)`	$[\![op]\!]([e]_\rho)$
t[p]	`Array val(t,p)`	$\tau(\mathrm{N}v)$ si $\rho(t) = \texttt{array}(\mathrm{N}, \tau)$ et $[p]_\rho = v$ et que v est un `int`
t.size	`Size_tab(t)`	$\tau(\mathrm{Nsize})$ si $\rho(t) = \texttt{array}(\mathrm{N}, \tau)$
f(args)	`Func(f,args)`	Soit $\text{params}, \text{body} = \rho_F(f)$. Soit $M_2, \rho_2 = \tau(\texttt{rev}(\text{params}), \texttt{rev}(\text{args}), \rho, M, \emptyset)$. Soit $M_3, \rho_3 = [\![\text{body}]\!](M_2, \rho_2)$. Le résultat est $\rho_3(\#\text{result})$. La mémoire finale est M_3, ρ.

FIGURE 2.6 – Syntaxe et sémantique (valeur) de *expr*

Texte	OCaml	Sémantique : $[\![instr]\!](\rho)$
x := e ;	`Affect(x,e)`	$\rho[x \leftarrow [e]_\rho]$
{ l }	`Block(l)`	Si l est la liste $[i1; \cdots; ik]$, $[\![ik]\!](\cdots [\![i1]\!](\rho) \cdots)$.
if test then i1 else i2	`IfThenElse(test,i1,i2)`	$\begin{cases} [\![i1]\!](\rho) \text{ si } [test]_\rho = true \\ [\![i2]\!](\rho) \text{ si } [test]_\rho = false \\ \text{non-défini sinon} \end{cases}$
while test body	`While(test,body)`	$\begin{cases} [\![\texttt{While}(test, body)]\!]([\![body]\!](\rho)) \\ \text{ si } [test]_\rho = true \\ \rho \text{ si } [test]_\rho = false \\ \text{non-défini sinon} \end{cases}$
t[p] := e ;	`Affect_array(t,p,e)`	$(M[\tau(\mathrm{N}v) \leftarrow [e]_\rho], \rho)$, si $\rho(t) = \texttt{array}(\mathrm{N}, \tau)$ et $[p]_\rho = v$
basic t[s] ;	`Array_decl(basic,t,s)`	$\rho[\mathrm{t} \leftarrow array(\mathrm{t}\#, \rho)][\mathrm{t}\#\mathrm{size} \leftarrow v]$ si $[s]_\rho = v$ Réalise une allocation (en assembleur).
f(args) ;	`Proc(f,args)`	Soit $\mathrm{params}, \mathrm{body} = \rho_F(f)$. Soit $M_2, \rho_2 = \tau(\texttt{rev}(\mathrm{params}), \texttt{rev}(\mathrm{args}), \rho, M, \emptyset)$. Soit $M_3, \rho_3 = [\![\mathrm{body}]\!](M_2, \rho_2)$. On renvoie M_3, ρ.
return ;	`Return None`	ρ. Termine l'exécution de la fonction courante (dans le code, exception).
return e ;	`Return (Some e)`	$\rho[\#\mathrm{result} \leftarrow [e]_\rho]$. Termine l'exécution de la fonction courante (dans le code, exception).
print s ;	`Print_str(s)`	ρ (provoque un affichage).
print e ;	`Print_expr(e)`	ρ, après avoir calculé $[e]_\rho$ (provoque son affichage).
basic x ;	`Decl(basic,x)`	ρ (aucun effet, ne sert qu'à l'analyse de type).

FIGURE 2.7 – Syntaxe et sémantique de *instruction*

ailleurs, elle impose de ne passer que des cases mémoires comme paramètres, et non des expressions complexes.

Ainsi, dans des langages impératifs n'ayant aucune notion de pointeurs, c'est la seule manière de simuler des effets de bords, et en général est donc présente à coté d'un appel par valeur (comme dans notre langage ou en Pascal, même si ce dernier dispose également de pointeurs), et nécessite donc que la fonction traite différemment les passages par valeur et par référence.

Peu de langages gardent aujourd'hui cette technique.

Pointeurs et passage par valeur En C, la seule stratégie de passage de paramètre est celle par valeur. D'ailleurs, vous savez que lorsqu'on définie une fonction, on ne précise rien sur les paramètres, hormis leur type (à la différence de ce qu'on fait ici). Au contraire, si on veut qu'une variable subisse des effets de bords, alors la solution est d'utiliser des pointeurs. Ainsi, la fonction appelante aura un pointeur passé par valeur (qu'elle ne pourra pas modifier), mais les effets de bords qu'elle effectuera sur la valeur pointée seront bien visible par la fonction appelante (puisque la zone pointée est la même dans les deux fonctions).

En fait, en présence de pointeurs, c'est une stratégie commune de ne mettre que de l'appel par valeur et d'utiliser les pointeurs pour simuler un passage par référence. Cela permet de simplifier le traitement de l'appel de fonction par le compilateur.

D'ailleurs, c'est en partie un comportement qu'on a déjà dans notre langage, quand on regarde les tableaux : les tableaux sont en réalité modélisés par des pointeurs, et même si on les passe par valeur, les effets de bords sont bien visibles par l'appelant.

Dans des langages objets, ou avec des types de données avec masquage, on a un comportement plus complexe – mais similaire – appelé parfois call-by-sharing. Cela consiste techniquement en le passage par valeur d'une référence vers une valeur complexe, dont les effets de bords de la fonction appelée seront visibles par la fonction appelante, mais dans laquelle la fonction appelante n'a accès qu'à une partie des information de la valeur (ses informations publiques). Typiquement, c'est ce qui se passe avec les objets en java (bien que java n'utilise pas ce terme).

Au final, toutes ces subtilités n'ont un effet qu'en présence d'effets de bords. Sur des langages purement fonctionnels (c'est-à-dire sans aucun effet de bord), toutes ces stratégies sont équivalentes.

Passage par copy-restore et aliasing C'est une variante du passage par référence : au lieu de passer une référence vers la valeur d'origine, on créée une copie de la variable, on y applique les effets de bords, et une fois l'appel terminé, on copie le contenu de la copie dans l'ancienne variable de l'appelant.

Dans des programmes séquentiels, le comportement du copy-restore est le même que le passage par référence excepté lorsqu'une même variable est passée comme paramètre d'une fonction (cas appelé aliasing).

Dans ce dernier cas, dans le cas d'un passage par référence la variable subit les effets de bords des 2 arguments auxquels elle est passée (menant à des effets parfois imprévisible). Dans le cas d'un passage par copy-restore, la fonction se comporte comme si elle avait deux variables différentes contenant la même valeur à l'appel de la fonction. À la fin de l'appel, la variable prendra la valeur de l'une de ses deux copies – et dans ce cas, c'est l'ordre des restaurations qui fixera lequels des deux arguments aura son résultat copié.

Dans tous les cas, le comportement de l'aliasing est peu aisé à manipuler, et il faut essayer de n'y recourir que lorsque c'est vraiment ce qu'on recherche, en connaissant bien le comportement du langage qu'on utilise.

Si on considère par exemple la fonction suivante :

```
null f(var int x, var int y) ::={
    x := x+1;
    y := y-1;
}
```

Dans un passage par référence classique (i.e., dans notre langage), l'appel de `f(a,a)` laissera la valeur de `a` inchangée.

Dans le cas d'un passage par copy-restore, si on restore les arguments de gauche à droite (`x` puis `y`), alors `a` verra sa valeur diminuée de 1, alors que si on les restore de la droite vers la gauche, il verra sa valeur augmenter de 1.

On peut même avoir des cas où le comportement est encore plus différent :

```
null g(var int x, var int y) ::={
    x := x+1;
    y := y/(x-y);
}
```

Dans le cas d'un passage par référence, l'appel de `f(a,a)` produira une erreur (division par 0), alors qu'avec un copy-restore, après l'appel, `a` sera soit augmentée de 1 (valeur de `x`), soit inchangée (valeur de `y`).

Stratégies non-strictes : Passage par nom et variantes

Contrairement au passage par valeur, on n'évalue pas les paramètres avant d'appeler la fonction sur le résultat du calcul, mais on remplace chaque occurence de l'argument par le paramètre dans le code de la fonction et on n'évalue le paramètre que lorsqu'il est utilisé.

Dans le call-by-name originel, cela est fait en conservant le contexte d'appel de la fonction de manière à ce que la valeur du paramètre soit le même que si on l'avait évalué à l'appel de la fonction.

Cette idée vient d'un formalisme nommée le lambda-calcul (qu'on ne détaillera pas ici) qui au lieu de considérer une notion de programme basé sur une machine abstraite (comme dans ce cours) considère une notion de ré-écriture de programme – et dans lequel l'appel de fonction se modélise naturellement par ce concept de call-by-name (on remplace le nom de la fonction par son corps, dans lequel chaque argument est remplacé par les paramètres).

L'avantage de cette stratégie est que si un paramètre n'est pas utilisé, alors il ne sera jamais évalué (ce qui est particulièrement avantageux si le paramètre en question ne termine pas ou renvoie une erreur).

Le désavantage assez évident est que si un paramètre est utilisé plusieurs fois, alors il sera évalué plusieurs fois, ce qui peut évidemment être coûteux en pratique.

En terme de compilation, ce type de stratégie demande également des structures plus complexes : là où dans l'évaluation stricte il est suffisant de stocker le résultat (ou une adresse) sur la pile, dans une stratégie non-stricte, il faut stocker le terme entier, ansi que son contexte, afin de réaliser son évaluation correctement même lorsque le contexte de la fonction appelée est différent. Cela demande de stocker plus d'informations en mémoire.

Par ailleurs, en cas d'effet de bord de ce calcul, cela causera plusieurs fois ces effets de bords. En fait, ce type de stratégie est considéré dans deux types de cas : dans des langages purement fonctionnels (type Haskell), où il n'y a jamais d'effets de bords (et ce n'est pas un problème), où comme stratégie explicite de certains langages où c'est très exactement le but recherché (typiquement en java, en fournissant un supplier à une lambda).

Évaluation paresseuse Ce désavantage est tellement flagrant que dans les langages où c'est la norme, on a une variante qui évite ce désavantage, l'évaluation paresseuse (ou call-by-need). Cette variante consiste simplement à ajouter de la mémoïsation au procédé en calculant un paramètre lors de la première fois qu'on le nécessite, et en stockant le résultat de manière à le réutiliser si c'est à nouveau nécessaire. Cela garde l'avantage de n'évaluer un paramètre que si nécessaire en éliminant totalement le désavantage, avec tout de même un coût en mémoire légèrement supérieur (puisqu'il faut se souvenir du paramètre et de son résultat). Haskell (qui est un langage purement fonctionnel) utilise ce comportement par défaut. En OCaml, ce n'est pas le comportement par défaut (OCaml utilise du passage par valeur), mais il est possible de le simuler en utilisant le module Lazy.

Macros Une autre variante du call-by-name sont les macros (par exemples en C++). Dans celles-ci, on remplace les arguments par les paramètres syntaxiquement, et non en gardant le contexte d'appel. Cela fait qu'un même paramètre pourra avoir des valeurs différentes à plusieurs évaluations dans la macro, ce qui est une source de bug potentiel. Mais cela est plus simple à implémenter pour le compilateur (particulièrement dans des langages qui ne sont pas fonctionnels).

Chapitre 3

Analyse Syntaxique

3.1 Introduction

Au chapitre 2, on a définit un programme comme un objet abstrait, structuré avec une forme arborescente qui suffit à en donner une sémantique claire. Il ne vous aura cependant pas échappé qu'en pratique, on ne code pas de cette manière-là (et que d'ailleurs ce serait humainement peu lisible). Au lieu de cela les programmes sont stockés sous la forme de fichier textes représentant le programme, mais en n'ayant pas la structure d'arbre clairement apparente. Passer d'une représentation telle que dans le chapitre 2 à une représentation usuelle est assez aisé (vous pouvez d'ailleurs voir comment le faire dans le module `Ast` du TD sur l'interpréteur). Cependant, un compilateur doit faire exactement la tâche inverse, et cette tâche est bien moins aisée. Dans ce chapitre, nous allons explorer la théorie et les outils permettant de réaliser cette tâche.

On a déjà mentionné que la définition d'un langage de programmation était en réalité une grammaire algébrique, mais qu'on s'intéressait aux arbres de dérivation et non aux langages générés. En fait, en allant plus loin, la description d'un programme sous forme texte peut être obtenue à partir de la même grammaire, mais en s'intéressant cette fois au mot généré, qui sera alors la description textuelle du programme qui est l'arbre de dérivation.

Formalisons pour commencer le problème que nous avons à résoudre : l'analyse syntaxique, ou parsing. En cours de Modèle de la Programmation et du Calcul, on a caractérisé les langages de grammaires algébriques en se demandant pour une grammaire donnée quels étaient les mots engendrés. Cependant, en compilation, ce qui nous intéresse, ce n'est pas de dire «oui, ce programme est syntaxiquement correct», mais de récupérer ce programme (c'est-à-dire son arbre syntaxique). C'est ce problème qu'on appelle l'analyse syntaxique :

Définition 3.1.1. *Problème de l'analyse syntaxique*

Entrée : *Une grammaire algébrique G et un texte (mot) w.*

Sortie : *Si G peut générer w, renvoyer un arbre de dérivation de G générant w.*

Un tel problème est complexe et nécessite des outils de théorie des langages pour être résolu proprement.

Les grammaires définissant les programmes ont pour non-terminaux non pas des lettres, mais des éléments abstraits qui sont les éléments de base du programme (nombres, identifiants, mots-clefs, etc). Si on cherchait à faire tout d'un coup, il faudrait alors modifier la grammaire pour qu'elle génère du texte, et elle serait plus grosse et moins lisible. Par

ailleurs, il est simple de voir que si pour un programme, il est nécessaire d'avoir une grammaire algébrique, les éléments de base sont bien plus simples et sont en réalités des mots de langages réguliers (un nombre est une suite de chiffres, par exemple).

Partant de ce fait, on peut découper cette tâche en deux phase : l'analyse lexicale (ou lexing) et l'analyse syntaxique (ou parsing) :

— L'analyse lexicale consiste à reconnaître les éléments de bases et à transformer une chaîne de caractères en liste d'éléments de bases.
— L'analyse syntaxique consiste à construire un arbre de dérivation générant la liste d'éléments de bases produite par l'analyse lexicale.

Théoriquement, on pourrait laisser le programmeur les réaliser à la main, mais cela lui demanderait une bonne connaissance de la théorie et serait fastidieux. Heureusement, on dispose d'outils automatiques pour aider à cela : les générateurs de lexeurs et les générateurs de parseurs. Nous en utiliserons deux dans le cadre de ce chapitre : Ocamllex pour le premier, et Menhir pour le second.

Dans ce qui suit, nous allons présenter plus formellement les deux problèmes, les techniques qui existent pour y répondre (et le lien avec ce que vous avez vus en Modèle de la Programmation et du Calcul), ainsi que les particularités liées à la tâche de compilation.

3.2 Analyse Lexicale

Cette section est dédié à décrire le principe du lexing et l'algorithme utilisé pour le réaliser.

3.2.1 Principe

Formellement, on commence par définir un ensemble de tokens qui est un type fini, c'est-à-dire une énumération, et tel que chaque élément de cet ensemble peut contenir une donnée d'un type quelconque. Par exemple, on peut considérer un sous-ensemble des tokens qu'on utilisera pour définir le langage de la section précédente suivant :

```
token ::= IF | ELSE | AFF | INT(int)| ID(string)| PLUS | MINUS | SEMICOL |
EQ
```

Ici, seuls les tokens `INT` et `ID` contiennent une données, les autres n'en contiennent pas.

Le but de l'analyse lexicale est de transformer un texte (séquence de caractères) en une séquence de tokens qu'il représente. Pour cela, il faut associer à chaque token l'ensemble des mots qui le représentent. Dans les langages de programmations, ces ensembles de mots sont toujours des expressions régulières, notion que vous avez croisée au semestre précédent. Techniquement, on pourra également définir des expressions qui ne représentent pas des tokens, mais doivent être ignorées (les espaces et les commentaires, typiquement).

Le lexing consiste alors à découper le textes en facteurs consécutifs dont chacun correspond à l'une des expressions régulières associées à un token ou à une séquence à ignorer, et à renvoyer la séquence de tokens correspondante ainsi produite. On peut donc voir le lexing comme une transduction [1] d'une séquence de caractères ASCII dans une séquence de tokens. Lorsque cela n'est pas possible (aucune séquence de tokens ne correspond au texte), le lexeur doit renvoyer une erreur.

Du moins, c'est là une vision un peu idéalisée qui correspond à ce qu'on cherche à faire en compilation, en pratique les lexeurs peuvent associer une action arbitraire à chaque

1. Terme d'informatique théorique pour décrire une fonction d'un langage dans un autre

expression régulière et donc réaliser un calcul quelconque guidé par l'analyse du texte au lieu de simplement renvoyer une séquence de token.

Si on reprend l'ensemble de tokens définis plus haut, sans donner tout de suite un lexeur adapté, on pourrait vouloir transformer le texte suivant :

if x = y - 4 then x := 42 + 5 else x := y - 9

en la séquence de tokens :

```
IF ID(x)EQ ID(y)MINUS INT(4)THEN ID(x)AFF INT(42)PLUS INT(5)
ELSE ID(x)AFF ID(y)MINUS INT(9)
```

3.2.2 Expressions régulières

Les expressions régulières sont une manière de décrire les langages réguliers. Dans le cadre de la compilation on décrit des langages de chaînes de caractères. L'alphabet est donc l'ensemble des caractères (ASCII). Les expressions sont définies récursivement comme suit :

expr ::=	*c*	un caractère
	expr + *expr*	
	expr.*expr*	
	expr∗	

Vous avez vus en MPC le sens de ces constructions, on ne revient donc pas dessus ici. Néanmoins, dans les générateurs de lexeurs (et plus généralement dans tous les programmes qui manipulent des expressions régulières), on a une extension de cette syntaxe qui permet de décrire de manière plus concise les expressions régulières utiles. Ces extensions sont uniquement du sucre syntaxique et pourraient être décrites dans le formalisme ci-dessus. On en donne la syntaxe utilisée en OCamllex (proche, mais pas identique à la notation POSIX) :

expr ::=	`'c'`	un caractère
	`_`	n'importe quel caractère
	`"s"`	une chaîne de caractères
	`['c1' 'c2' ... 'ck']`	l'un des caractères listés
	`['c1' - 'c2']`	l'un des caractères entres c1 et c2
	`[^ 'c1' 'c2' ... 'ck']`	un caractères qui n'est pas listé
	`(expr1 | expr2 | ... | exprk)`	l'union
	`expr1 expr2`	la concaténation
	`expr*`	itération de l'expression
	`expr+`	itération non-vide de l'expression
	`expr?`	une chaîne de l'expression ou le mot vide

Si on reprend le langage du cours, voici quelques exemples :

- `IF` correspond à `"if"`,
- `INT` correspond à `['0' - '9']+`,
- `ID` à `['a' - 'z' 'A' - 'Z'] ['a' - 'z' 'A' - 'Z' '0' - '9' '_']*`.

Il existe d'autres programmes qui utilisent des expressions régulières. La syntaxe en est similaire (mais possiblement légèrement différente). Par exemple, on peut citer des programmes comme `sed` ou `grep` qui les utilisent.

3.2.3 Lien avec la théorie des langages

En Modèles de la Programmation et du Calcul, vous avez vu la notion d'automates finis et vus que c'est une autre manière de décrire les langages décrits par les expressions régulières, dit rationnels ou réguliers. Les automates sont en réalité des programmes (et il est relativement aisé de produire un code simulant un automate déterministe). Comme vous l'avez vu, il existe des techniques pour transformer une expression régulière en un automate (Glushkov ou Thompson, par exemple), et il est possible de déterminiser des automates. On ne rappellera pas ici ces techniques (reportez-vous à votre cours de MPC). On va plutôt s'intéresser à leur application à l'analyse lexicale.

Il y a une différence importante entre les automates tels que vous les avez vus et l'analyse lexicale : les automates déterminent si un mot appartient ou non à un langage régulier, l'analyse lexicale quant à elle cherche à découper un mot en facteurs appartenant à des langages réguliers et produire la liste des tokens correspondant.

Il convient donc d'expliquer comment se servir d'un automate pour faire cette tâche. Cela va consister à donner une autre signification au calcul d'un automate, c'est-à-dire, une sémantique. Oui c'est le même mot qu'au chapitre 2, et non, ce n'est pas un hasard. En effet, comme dit plus haut, un automate est un programme (dans un langage de programmation un peu particulier, certes) et il est donc possible de lui définir un sémantique. En MPC, la sémantique était une fonction des mots vers les booléens. En compilation, ce sera une fonction des mots vers des séquences de tokens (idéalement – en réalité vers des séquences d'actions).

On rappelle d'abord la notion d'automate déterministe.

Définition 3.2.1. *Automate déterministe*

Un automate déterministe est un tuple $\mathcal{A} =< Q, i, F, \Delta >$ *avec :*

— Q *un ensemble d'états fini.*
— $q_i \in Q$ *un état initial.*
— $F \subseteq Q$ *un ensemble d'états finaux.*
— $\delta : (Q \times \Sigma) \to Q$ *une fonction de transition.*

On ne considèrera que les automates déterministes, bien qu'on pourrait définir les calculs sur des automates non-déterministe (ce qui reviendrait à exécuter à la volée son déterminisé, ce qui est possible mais qu'on choisit de ne pas aborder ici).

Une configuration de l'automate est un tuple (q, l, f, i), où $q \in Q$ est l'état courant, l est la longueur du mot lu, $f \in F \uplus \{\bot\}$ est le dernier état final rencontré ($\bot$ s'il n'y en a pas), et i est la position du dernier état rencontré.

Le calcul de $\mathcal{A}$ sur u, $\mathcal{A}(u)$ est défini récursivement par :

— $\mathcal{A}(\varepsilon) = (q_i, 0, \bot, 0)$.
— $\mathcal{A}(ua) = (q', l+1, f, i)$ si $\mathcal{A}(u) = (q, l, f, i)$, qu'il existe une transition (q, a, q'), et que q' n'est pas final.
— $\mathcal{A}(ua) = (q', l+1, q', l+1)$ si $\mathcal{A}(u) = (q, l, f, i)$, qu'il existe une transition (q, a, q'), et que q' est final.
— $\mathcal{A}(ua) = (\bot, l, i, f)$ si $\mathcal{A}(u) = (q, l, f, i)$ et qu'il n'existe pas de transition étiquetée par a depuis q.

Observez qu'une fois qu'une configuration de la forme $(\bot, l, i, f)$ est atteint, quelques soient les lettres lues, la configuration ne changera pas, et en pratique, on arrête le calcul à ce moment-là.

Ce calcul permet de déterminer le plus long préfixe d'un mot qui est accepté par $\mathcal{A}$ au sens habituel. Ce qui tombe bien, puisque c'est exactement ce qu'on cherche à faire dans l'analyse syntaxique.

Les seules valeurs utiles dans ce cadre une fois le calcul terminé sont les deux derniers éléments de la configuration, à savoir l'état final atteint et longueur du mot ainsi reconnu. Si $\mathcal{A}(u)$ est (q, l, f, i), alors on appelle $\text{fin}(\mathcal{A}, u) = f$ et $\text{pos}(\mathcal{A}, u) = i$.

L'analyse lexicale d'un mot u par un automate déterministe consiste à exécuter $\mathcal{A}$ sur u, puis à le rappeler sur $u_{\text{pos}(\mathcal{A},u)\cdots|u|-1}$, c'est-à-dire u auquel on a retiré $u_{0\cdots\text{pos}(\mathcal{A},u)-1}$, et considérer la séquence d'états finaux ainsi générée. Formellement, on peut définir cela récursivement par

— $\text{token}_{\mathcal{A}}(\varepsilon) = \varepsilon$

— $\text{token}_{\mathcal{A}}(u) = \text{fin}(\mathcal{A}, u), \text{token}_{\mathcal{A}}(u_{\text{pos}(\mathcal{A},u)\cdots|u|})$

En pratique, chaque état final peut être associé à du code arbitraire. Dans le cas d'un compilateur, les actions seront au choix :

— Un token,

— Une erreur (auquel cas l'analyse s'arrête)

— Ignorer (auquel cas, on passe au suffixe suivant). Techniquement, ce cas se code en pratique par un appel récursif au lexeur, mais dans ce formalisme, on peut ignorer (ou de manière équivalente, considérer qu'il existe un token spécial qu'on efface).

Avec ces actions, si aucune erreur n'est déclenchée, l'analyse d'un mot produira bien une liste de tokens, ce que l'on cherche bien à obtenir.

3.2.4 Application au problème de l'analyse lexicale

Dans la section précédente, on est dans un cas idéalisé où on a un seul automate, et où on n'a pas précisé ce qu'on fait de l'information de sur quel état final on termine. Dans la pratique, pour réaliser une analyse lexicale, on va avoir plusieurs tokens à reconnaître (ou plus généralement, plusieurs cas à considérer) qui vont tous être associés à une expression régulière.

Le problème principal est que chaque chaîne de caractères doit correspondre à un seul cas. Cependant, en pratique, forcer l'utilisateur à donner des expressions régulières disjointes serait peu commode (et peu maintenable). Par exemple, dans notre langage, on aura une expression régulière associée au token `ID` qui capture toutes les chaînes de caractères composée de caractères alphanumériques, et une autre associée au token `IF` qui est simplement la chaîne «if». Évidemment, «if» est également capturé par l'expression que l'on souhaite associée à `ID`, et préciser qu'on n'a pas ce cas serait fastidieux (d'autant qu'il faudrait alors le faire pour tous les tokens où c'est le cas, et il y en a un certain nombre).

La solution proposée par les générateurs de lexeurs est que les règles les plus hautes dans un fichier de description de lexeur sont prioritaires par rapport aux plus basses. Évidemment, cela veut dire que dans la traduction de ces expressions régulières en automates, l'algorithme tient compte de cet ordre de priorité, et en particulier que si un état final correspond à plusieurs expressions régulières, on le fera correspondre à celle de plus haute priorités parmi celles-ci.

Le second problème est que certaines chaînes de caractères doivent être reconnues pour être ignorées. Typiquement ce sera le cas des caractères d'espacement (dans de nombreux langages - mais pas tous), et des commentaires. Ces expressions ne doivent donc pas être associés à des tokens (bien qu'on pourrait avoir un token particulier ignoré par la suite du processus). Ce problème est réglé par le fait qu'en réalité chaque expression est associée à une action arbitraire, et en particulier il est tout à fait possible de rappeler récursivement le lexeur sur la suite de la chaîne à lexer, ce qui est ce qui se passe en pratique.

Cette capacité à appliquer des actions quelconque va en réalité être utile pour un deuxième cas d'utilisation pratique : certains tokens, typiquement ceux représentant les

constantes et les identifiants du langage ont besoin d'une information supplémentaire que le token lui-même (à savoir la valeur en question, l'entier 1 et l'entier 42, c'est pas pareil). Pour cela, l'action associée à l'expression va récupérer la chaîne de caractères analysée et calculer la valeur associée.

Dernier point important : on a expliqué dans la partie théorique que les lexeurs cherchent le *plus long* préfixe accepté par l'une des expressions régulières. Ce choix qui peut paraître surprenant ne l'est pas tant que ça si on réfléchit au fait qu'une bonne partie des langages réguliers qui représentent les éléments lexicaux d'un langage de programmation sont clos par préfixe, et que par exemple, 123 correspond à un entier, mais 12 et 1 également. Évidemment, quand vous écrivez `x := 123;` dans un code, vous voulez que ça corresponde à `VAR(x) ASSIGN INT(123)`, pas à `VAR(x) ASSIGN INT(1)INT(2)INT(3)`, ce qui est permis aisément par cette convention.

Cela permet également d'avoir des identifiant comme `ifvar` ou `while_var_if_true` qui sont reconnus comme des identifiants bien que l'expression correspondant aux identifiants soit située après celle sur les mots-clefs (et donc moins prioritaires). Cela donne aussi une bonne raison au refus d'avoir des espaces dans les noms d'identifiants : cela permet de distinguer simplement dans un texte `if var` (`IF ID(var)`) de `ifvar` (`ID(ifvar)`), et de plus, le choix inverse donnerait des codes ésotériques où `x := y+ x` n'a pas le même sens que `x:=y +x` (et je vous laisse juger de la lisibilité de ce choix).

C'est d'autant plus important de faire un choix que sinon il existe plusieurs décomposition possible d'un même texte en séquences de tokens, ce qui n'est évidemment pas acceptable (un compilateur doit être déterministe), sans parler du fait que ce nombre peut-être exponentiel (pour une séquence de lettres de taille n, il existe 2^n séquences de sous-séquences qui concaténées donnent cette séquence).

Ces choix permettent donc d'écrire des lexeurs utiles pour les langages de programmation avec une syntaxe relativement courte, puisqu'ils correspondent bien à leurs cas d'usages. Et ces conventions sont même pertinentes dans la plupart des cas d'usage des lexeurs.

Mentionnons enfin un point de complexité : en théorie, le procédé décrit plus haut est quadratique dans le pire des cas : en effet, le pire qui puisse se passer est que le texte doive être découpé caractère par caractère, mais que pour chaque découpage, le texte doive être lu en entier par l'automate avant de déterminé que le dernier état final lu l'a été avec la première lettre. En pratique, non seulement la plupart des expressions régulières apparaissant dans un langage de programmation sont closes par préfixes, mais très peu d'entre elles admettent des calculs qui peuvent avoir de longs chemins non-acceptant après avoir lu un mot accepté. La complexité observée réellement est donc dans la plupart des cas linéaire en la taille du texte d'entrée (ce qui est une bonne nouvelle).

3.2.5 Fichiers de lexeur en pratique avec Ocamllex

Vous aurez des exemples en TD, mais, en gros, un fichier se présente de la manière suivante :

- Une section de code OCaml quelconque (pour définir des valeurs communes et ouvrir des modules) qui sera copiée avant le code générée par le générateur. Elle est délimitée par des accolades.
- Une section pouvant contenir des définitions d'expressions régulières, telles que : `let digit = ['0' - '9']`
- Une section contenant la définition des règles de lexeurs. Une règle est donnée par un nom, commence par le mot-clé `parse` qui se note comme un pattern-matching

sur des expressions régulière de la manière suivante :

```
rule <name> = parse
     | <expr1> {<action1>}
     | <expr2> {<action2>}
```

Où les notation entre chevrons sont à remplacer. Toutes les «actions» sont des expressions OCaml qui doivent avoir le même type.

OCamllex génèrera alors un fichier OCaml avec une fonction par règle (de même nom) qui prend en argument un paramètre de type `Lexing.lexbuf` (qui est un buffer sur la chaîne où le fichier à analyser) et renvoie un résultat ayant pour type celui des actions (typiquement un token).

Pour l'utiliser, on initialise un paramètre de type `Lexing.lexbuf` avec les fonctions de la bibliothèque `Lexing` et on appelle la fonction correspondant à la règle que l'on souhaite appliquer.

Un appel à la règle va déterminer le plus long préfixe de la chaîne dans le `lexbuf` et :

— renvoyer le résultat correspondant à la première expression qui lui correspond (de haut en bas).

— placer le buffer sur la position suivant directement ce préfixe.

Pour lexer un fichier ou une chaîne entière, il suffit donc de rappeler récursivement la règle sur le même buffer jusqu'à recevoir un token caractéristique de la fin du fichier.

Il existe bien sûr d'autres générateurs de lexeurs dans d'autres langages (`JFlex` pour Java, `lex` puis `flex` pour C, et bien d'autres), qui ont en général un fonctionnement et une syntaxe très similaire à OCamllex (en partie parce que tous les outils cités ici ont été inspirés par `lex`).

3.2.6 Fonctionnement (résumé) d'un générateur de lexeur

Un générateur de lexeur suit globalement les étapes suivantes pour générer le code exécutable correspondant à un fichier de lexeur :

— Pour chaque expression, créer un automate qui lui correspond (avec Thompson ou Glushkov – en pratique, Thompson est souvent utilisé). Éventuellement (avec Thompson notamment) simplifier l'automate (pour éliminer les ε-transitions).

— Fusionner ces automates en tenant compte des règles de priorités.

— Déterminiser puis minimiser l'automate obtenu, et l'écrire dans une «table» de lexing (qui est un format compact de représentation d'automate). Ce point est facultatif, on peut directement faire le point suivant sur des automates non-déterministes (compromis différent de coût).

— Écrire un code (générique) qui lit une table en parallèle d'un buffer pour simuler l'exécution telle que décrite plus haut et une fois l'état final déterminé, exécuter l'action correspondante.

Cela est valable pour la plupart des générateurs de lexeurs.

Il existe cependant des programmes ressemblant fortement à des lexeurs, mais qui, pour des raisons de cas d'utilisation ne font pas tout à fait les même choix. En particulier, c'est le cas de `sed` qui prend en argument deux expressions régulières et remplace les occurences de la première par la seconde dans un texte. Dans ce programme, on n'utilise qu'une fois les expressions régulières, auquel cas, le coût de déterminisation est grand pour l'utilisation qu'on en fait, alors qu'il est possible en pratique d'exécuter un automate non-déterministe de la même manière mais sans le déterminiser au préalable (la déterminisation se fait à la volée). Cela se fait au coût (dans le pire des cas, rarement atteint en pratique) d'un espace exponentiel.

Dans un compilateur en revanche, on peut avoir à gagner à déterminiser puis minimiser l'automate : on ne fera cette opération qu'une fois, alors que le lexeur sera utilisé de nombreuses fois. Dans la plupart des cas, l'automate obtenu n'est pas de taille exponentielle, et on obtient donc un lexeur de taille comparable au non-déterministe avec une complexité mémoire constante.

Dans le TD associé à ce chapitre, pour clarifier le fonctionnement d'un lexeur et illustrer le fait qu'on exécute bel et bien un automate, on implémentera un mini lexeur générique qui prend en entrée un automate et l'exécute sur un mot avec la sémantique décrite plus haut. On n'implémentera pas la traduction des expressions régulière vers les automates (ce serait faisable, mais ça nous emmènerait un peu loin).

3.2.7 Lexeur du langage du cours

En TD, vous aurez entre plus à écrire un lexeur pour le langage du cours, et à l'utiliser pour divers buts.

On donne ici les tokens correspondant aux éléments syntaxiques du cours (les commentaires sont là pour expliciter les tokens dont la signification du nom n'est pas immédiate) :

```
type token =
| IF
| THEN
| ELSE
| WHILE
| L_PAR (* ( *)
| R_PAR (* ) *)
| L_CUR_BRK (* { *)
| R_CUR_BRK (* } *)
| L_SQ_BRK (* [ *)
| R_SQ_BRK (* ] *)
| ADD
| SUB
| MUL
| DIV
| MOD
| AND
| OR
| NOT
| EQ
| NEQ
| LT
| GT
| LEQ
| GEQ
| COMMA
| SEMICOLON
| ASSIGN
| DEF
| DOT
| PRINT
| SIZE
```

```
| INT_TYP
| FLOAT_TYP
| BOOL_TYP
| VAR
| ID of string
| STRING of string
| INT of int
| FLOAT of float
| BOOL of bool
| EOF
```

3.3 Parsing

3.3.1 Principe et rappel de grammaires algébriques

Une grammaire algébrique est un formalisme permettant de décrire des langages dits algébriques, qui contiennent strictement les langages réguliers.

On rappelle la définition d'une grammaire :

Définition 3.3.1. *Un grammaire algébrique est un tuple $G =< \mathcal{N}, \mathcal{T}, S, R >$ où :*

- *$\mathcal{N}$ est un ensemble fini de non-terminaux*
- *$\mathcal{T}$ est un ensemble fini de terminaux*
- *$S \in \mathcal{N}$ est le non-terminal initial*
- *$R \subseteq \mathcal{N} \times (\mathcal{N} \uplus \mathcal{T})^*$ est un ensemble de règles de dérivation.*

L'application d'une règle $r = N \to w$ à un mot $u = u_0 N u_1$ donne le mot $u' = u_0 w u_1$, et se note $u \underset{r}{\to} v$.

On peut dériver un mot v depuis un mot u s'il existe une séquence de règles qui permet de transformer u en v. On le note $u \underset{G}{\overset{*}{\to}} v$.

Le langage d'une grammaire est l'ensemble des mots de $\mathcal{T}^*$ dérivables depuis S : $L(G) = \{w \in \mathcal{T}^* \mid S \underset{G}{\overset{*}{\to}} w\}$.

Si l'on regarde attentivement le langage défini au chapitre 2, on peut s'apercevoir aisément qu'il s'agit en réalité d'une grammaire algébrique dont les non-terminaux sont `binop`, `unop`, `expr`, `instr`, `arg` et `declFunc`, ainsi que `arg_list` et `instr_list`, le reste (i.e., les mots clefs et constantes du langages) en étant les terminaux. Les définition inductives des expressions, instructions (et le reste) sont les règles de dérivations de cette grammaire.

Si on garde cette grammaire telle quelle, on produira des programmes écrits tels que dans le code de l'interpréteur codé en TD. Cette écriture n'est pas pratique, et l'on changera donc l'écriture de ces règles (mais pas les règles elles-mêmes) pour générer finalement un langage de programmation plus habituel (correspondant à l'affichage qui est fait de ces programmes par l'interpréteur). Cela signifie, par exemple, qu'au lieu d'avoir la règle

$$expr \to \texttt{IfThenElse}(expr, instr, instr)$$

on considèrera la règle

$$expr \to \texttt{IF}\ expr\ \texttt{THEN}\ instr\ \texttt{ELSE}\ instr$$

La grammaire génèrera alors de séquences de tokens (vus dans la partie lexing), qui seront alors nos terminaux.

Lié au lexeur, cela nous permet de définir l'ensemble des chaînes de caractères représentant un programme syntaxiquement correct de notre langage de programmation.

Cependant, le but de la phase d'analyse syntaxique du compilateur est de faire l'inverse, à savoir :

Entrée : Une séquence de tokens

Sortie : Un programme (au sens du chapitre 2) lui correspondant.

On notera que les deux objets sont en réalité définis par la même grammaire, à la structure des productions des règles près, mais les règles elles-mêmes sont les mêmes. Donc, si étant donné une séquence de tokens, on est capable de déterminer une dérivation la produisant, on pourra en déduire le programme correspondant (en les réappliquant, mais en générant la forme du chapitre 2).

En pratique, comme pour le lexeur, on pourra associer des actions arbitraire à chaque règle de dérivation, mais dans le cadre de la compilation, on se contentera en général de produire pour chaque règle le programme (ou expression ou instruction) correspondant au sous-mot reconnu.

Cependant, c'est là un problème non-trivial (beaucoup moins que dans le cas du lexing), et d'ailleurs pas possible de manière déterministe pour toutes les grammaires. Avant de voir comment nous y prendre théoriquement, commençons par expliquer rapidement comment écrire une grammaire pour l'outil OCaml, menhir.

3.3.2 Générateurs de parseurs

On présente ici la forme d'un fichier de grammaire accepté par menhir. Ce format de fichier est hérité d'autres outils, tel que yacc, et donc, à quelques adaptations près, la plupart des outils répandus pour générer des parseurs pour d'autres langages auront une syntaxe et une organisation similaire. La forme générique d'un fichier est la suivante.

```
{%
    <code OCaml>
%}

<declarations de tokens>
%token EOF
%token <int> INT

<priorites et associativite -- cf plus bas>

%start <type> main

%%

<regles de derivations>
main:
| regle1     {expression1}
| regle2     {expression2}

{%
    <code OCaml>
%}
```

La première partie du fichier, entre `{%` et `%}`, contient du code OCaml. Ce code copié tel quel au début du fichier .ml généré par menhir. Elle permet entre autre de déclarer des fonctions qui seront utilisées dans les règles du parseurs. Il est possible de mettre un autre bloc de code en fin de fichier, qui sera copié à la fin du parseur généré – et qui donc ne peut être utilisé par le parseur, mais peut utiliser le parseur.

Ensuite, avant la ligne `%%`, se situent les déclaration des terminaux (tokens) et non-terminaux.

Les tokens se déclarent avec le mot clef `%token` et sont constitués d'un constructeur pouvant (ou non) contenir des données (comme un type collection usuel). Le type contenu se met entre `< >`.

Le non-terminal initial se déclare avec le mot-clef `%start` et doit être typé avec la même syntaxe que les tokens.

Les autres non-terminaux peuvent être déclarés et typés avec le mot clef `%type`. Cette déclaration est facultative si on utilise l'inférence de type d'OCaml (il faut passer la bonne option à menhir pour cela).

Il y a d'autres déclarations possibles dans cette section, notamment les déclarations de priorité et d'associativité qui permettent de résoudre automatiquement des conflits que l'on explique à la section 3.3.10.

Après la ligne `%%` on place la grammaire proprement dite. Celle-ci est constituée d'une liste de règles auxquelles on associe une action. La forme générale est

```
non-terminal:
| regle1 {expression1}
| regle2 {expression2}
```

Une règle est dans sa forme la plus simple une séquence de terminaux et de non-terminaux. On peut nommer la valeur contenue dans l'un des composant de la règle avec la syntaxe `nom = letter` où `nom` est un nom quelconque et `letter` le terminal ou non-terminal dont on veut récupérer la valeur.

Toutes les expressions d'un même non-terminal doivent avoir le même type, mais en dehors de cette restriction on peut y mettre du code OCaml arbitraire.

Par exemple, voici une simple grammaire qui calcule des additions :

```
%token <int> INT
%token ADD
%start <int> expr

%%

expr:
| i = INT { i }
| v1 = expr ADD v2 = expr { v1 + v2 }
```

3.3.3 Ambiguïté

La principale difficulté avec les grammaires c'est qu'un mot peut être dérivé par plusieurs dérivation d'une même grammaire. Il faut cela dit distinguer deux cas.

Le premier est une fausse ambiguïté : si on prend la grammaire de notre langage il y a plusieurs dérivations permettant d'obtenir `WHILE BOOL(true) VAR(x) AFFECT INT(1)`

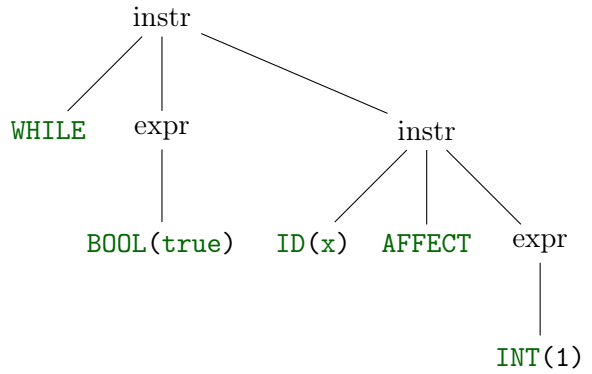
depuis instr. Par exemple :

$$\begin{aligned}
\textit{instr} &\rightarrow \texttt{WHILE}\ \textit{expr}\ \textit{instr} \\
&\rightarrow \texttt{WHILE}\ \texttt{BOOL(true)}\ \textit{instr} \\
&\rightarrow \texttt{WHILE}\ \texttt{BOOL(true)}\ \texttt{ID(x)}\ \texttt{AFFECT}\ \textit{expr} \\
&\rightarrow \texttt{WHILE}\ \texttt{BOOL(true)}\ \texttt{ID(x)}\ \texttt{AFFECT}\ \texttt{INT(1)}
\end{aligned}$$

$$\begin{aligned}
\textit{instr} &\rightarrow \texttt{WHILE}\ \textit{expr}\ \textit{instr} \\
&\rightarrow \texttt{WHILE}\ \textit{expr}\ \texttt{ID(x)}\ \texttt{AFFECT}\ \textit{expr} \\
&\rightarrow \texttt{WHILE}\ \texttt{BOOL(true)}\ \texttt{ID(x)}\ \texttt{AFFECT}\ \textit{expr} \\
&\rightarrow \texttt{WHILE}\ \texttt{BOOL(true)}\ \texttt{ID(x)}\ \texttt{AFFECT}\ \texttt{INT(1)}
\end{aligned}$$

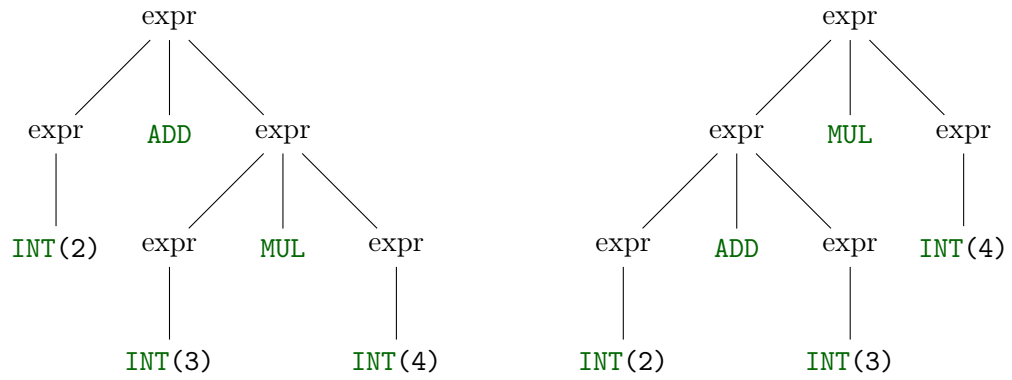
Ces deux dérivations sont valides et produisent la même séquence de tokens. Cependant, on voit qu'elles correspondent en fait au même programme et que c'est là une différence bénigne qu'on souhaite ignorer. Cette différence est dûe à l'ordre d'application entre elles de règles indépendantes, et ne peut donc pas affecter le terme produit.

La solution pour regrouper ces dérivations équivalentes est de regarder l'abre de dérivation, qui ici est le suivant



Cela permet de ne pas préciser l'ordre de dérivation qui est peu important.

Il existe cependant une réelle ambiguïté dans certaines grammaires qui est plus grave. Si dans notre grammaire, on considère les expressions arithmétiques et qu'on ne place pas les parenthèses, alors il existe deux arbres de dérivations différents pour la séquence `INT(2) ADD INT(3) MUL INT(4)` :



celui qui représentera l'expression `Binop(Add,2,Binop(Mul,3,4))` et celui qui représentera l'expression `Binop(Mul,Binop(Add,2,3),4)`. Ces deux expression n'ayant pas la

même valeur, un tel problème n'est pas acceptable dans un langage de programmation et il faudra utiliser des grammaires qui ne l'ont pas.

Une grammaire avec cette propriété est dite ambiguë, et un langage de programmation devra avoir une grammaire qui ne l'est pas. Et le truc, c'est que les grammaires naturelles pour les langages de programmation le sont, comme le montre l'exemple précédent (du moins à première vue).

Pour cet exemple, la solution la plus simple est de parenthéser toutes les expressions (ce qui est très désagréable pour le programmeur), mais une solution plus fine est d'avoir une grammaire qui respecte les règles de priorités usuelles et n'est pas ambiguë (ce qui est dur à déterminer).

Il existe un autre cas qui revient dans les langages de programmation, qui est celui du «dangling else». Dans la grammaire donnée au chapitre 2 il n'apparaît pas explicitement, mais on l'ajoutera pour la partie texte. Il se remarque sur la séquence de tokens suivante : `IF` expr `THEN IF` expr `THEN` instr `ELSE` instr. Cette séquence admet 2 interprétations, qui consistent à rattacher le `ELSE` au premier ou au second `IF`. La solution choisie est souvent de le rattacher au `THEN` le plus proche.

L'ambiguïté est un problème profond sur les langages, et étonnamment complexe à première vue. En particulier, étant donné une grammaire, il est indécidable en général de déterminer si cette grammaire est ambiguïté ou non. Il est également indécidable en général de produire une grammaire non-ambiguë équivalente à une grammaire donnée. Cependant, comme on va le voir dans la suite, il existe des techniques pour construire des algorithmes de parsing sur certaines sous-familles de grammaires, ainsi que de désambiguïser automatiquement des cas d'ambiguïté (qui sont fort heureusement ce dont a en général besoin un langage de programmation).

En particulier, pour ce qui nous intéresse, menhir (et d'autres outils aussi) permet de déclarer des règles d'associativité et de priorités des opérateurs qui permettent de parser de manière non-ambiguë des grammaire qui pourtant sont ambiguës (cf section 3.3.10). Et ça tombe bien, cette technique permet bien de désambiguïser les deux cas citer précédemment.

Si déterminer l'ambiguïté d'une grammaire est un problème indécidable, heureusement le problème de l'analyse syntaxique l'est, et il existe de nombreux algorithmes pour le résoudre. On en mentionnera certains, mais on ne développera vraiment que ceux qui sont utilisés en compilation aujourd'hui, en se concentrant réellement par celui utilisé par l'outil qu'on utilise en TD, menhir.

3.3.4 Algorithmes génériques – point historique

L'un des premiers algorithmes permettant de résoudre le problème du parsing est l'algorithme CYK (Cocke-Younger-Kasami, 1967) qui étant donné un mot et une grammaire produit tous les arbres de dérivations possibles pour ce mot.

Il consiste essentiellement à générer un tableau pour chaque couple de position qui pour chaque case (i, j) calcule les arbres de dérivation possible pour le mot commençant à la position i et terminant à la position j. Le calcul se fait récursivement (avec de la programmation dynamique).

L'algorithme fonctionne sur les grammaires en forme normale de Chomski (qu'on ne détaillera pas ici), mais toute grammaire peut être mise sous cette forme automatiquement.

Il n'est pas utilisé en compilation pour deux raisons principales :

— Il ne résout pas les cas d'ambiguïté, puisqu'il fonctionne sur les grammaire ambiguës, et ne permet donc pas de faire un choix entre les différentes dérivations possibles.

— Il nécessite une mémoire de taille quadratique et un temps cubique en la taille du mot analysé.

C'est évidemment le second point qui est fatal pour la compilation : les programmes peuvent être long, et une analyse qui aurait cette complexité ferait que sur des programmes conséquents, la compilation serait beaucoup trop longue.

Un autre algorithme historique est celui de Earley, datant de 1968. Tout comme l'algorithme CYK, il fonctionne sur des grammaires ambiguës et va permettre de déterminer tous les arbres de dérivations, et il en a une complexité proche (à savoir cubique en la taille de l'entrée). Cependant, il possède, contrairement à CYK une propriété qui est importante pour la compilation : il détecte les erreurs dès qu'elles surviennent, dans le sens où il respecte la propriété dite du préfixe valide. C'est à dire qu'à toute étape de l'algorithme, si on ne renvoie pas d'erreur, alors si on a lu le mot u, il existe un mot v tel que uv qui est engendré par la grammaire. Cette propriété est importante pour le retour d'erreurs, puisqu'elle permet de notifier la première erreur syntaxique survenant dans un code.

Ces deux algorithmes, et notamment le second, vont trouver une utilité cruciale en linguistique où les langages considérés sont intrinsèquement ambigus et où on a besoin de considérer tous les arbres possibles.

En compilation, pour des raisons d'efficacité et puisque les langages mis en jeu sont structurellement moins complexes que des langues naturelles, on va se tourner vers des algorithmes avec une meilleure complexité, mais qui ne fonctionnent pas sur toutes les grammaires.

L'idée de ces algorithmes sera d'analyser une séquence de tokens de gauche à droite en n'ayant pas besoin de retenir des informations sur tous les préfixes lus et en n'ayant jamais besoin de revenir en arrière. Ils auront une complexité en temps et en mémoire linéaire, ce qui pour les besoins de la compilation est nécessaire. Le prix à payer est qu'ils ne fonctionneront pas sur toutes les grammaires, et qu'ils pourront demander à l'utilisateur de modifier sa grammaire pour faire fonctionner l'algorithme.

Il y a deux approches principales : l'analyse descendante, qui consiste à construire un arbre de dérivation à partir de la racine en prédisant les règles à utiliser (par exemples les analyses LL), et l'analyse ascendante qui construit l'arbre à partir des feuilles. La plupart des outils utilisent cette deuxième approche sur laquelle nous nous concentrerons donc.

Toutes ces approches satisfont une propriété importante pour le parsing : celle du *préfixe valide*. Cette propriété dit qu'à tout moment dans l'algorithme de parsing, la chaîne lue jusqu'ici est le préfixe d'un mot engendré par la grammaire. Autrement dit, lorsqu'une erreur survient dans l'analyse, c'est au premier token qui empêche la chaîne lue d'appartenir au langage. Cela permet de retourner des erreurs plus compréhensibles pour l'utilisateur.

Pour obtenir ces algorithmes, on va comme pour le lexing utiliser des résultats de la théorie des langages, à savoir la notion d'automate à pile (adaptée).

3.3.5 Automates à pile shift-reduce

Les automates à pile sont une extension des automates avec une mémoire non-bornée qui prend la forme d'une pile. À chaque lettre lue, l'automate peut changer d'état, mais également soit empiler, soit dépiler un élément sur sa pile. La transition prise est de plus dépendante de l'élément présent au sommet de la pile.

Nous ne donnerons pas ici de définition formelle générale des automates à pile, pour nous concentrer sur la version utilisée pour le parsing (et ses variantes), que nous appellerons un automate à pile shift-reduce. On ne considèrera également que la version déterministe (puisque notre programme doit être déterministe).

Les plus curieux pourront se renseigner sur la notion générale d'automate à pile. La différence principale avec ce que nous présentons ici est que dans un automate à pile, l'alphabet de pile est quelconque, et la pile peut être modifée à chaque lecture de lettre par un empilement ou un dépilement.

Définition 3.3.2. *Automate à pile shift-reduce*

Un automate à pile shift-reduce est un tuple $\mathcal{A} =< Q, \Sigma, q_i, q_f, \lambda, \Delta >$, où :

- *Q est un ensemble d'états finis,*
- *Σ l'alphabet d'entrée découpé en $\mathcal{T}$ les terminaux et $\mathcal{N}$ les non-terminaux,*
- *$q_i \in Q$ est l'état initial,*
- *$q_f \in Q$ est l'état final,*
- *λ est une fonction de $Q \backslash \{q_i\}$ dans Σ qui associe à chaque état une lettre (celle qu'on doit lire pour l'atteindre, c'est pour ça que ça ne concerne pas l'état initial).*
- *Δ est l'ensemble des transitions qui peuvent être de deux formes :*
  - *(q, a, q') avec $q, q' \in Q$, $a \in \Sigma$, tels que $\lambda(q') = a$ sont les shifts,*
  - *$(q, a_1 \cdots a_k, n)$ avec $q \in Q$, $a_1, \cdots, a_k \in \Sigma$ et $n \in \mathcal{N}$ sont les reduce.*

Une configuration d'un automate à pile shift-reduce est une séquence non-vide d'états de Q et le mot restant à lire dans Σ^*, $c = (q_0 \cdots q_k, u)$.

La configuration initiale est $c = q_i, u$, où $u \in \mathcal{T}^*$ est le mot d'entrée.

Il y a une transition shift de c à c' en lisant a, noté $c \xrightarrow{\text{shift}(a)} c'$ si $c = q_0 \cdots q_k, au$, $c' = q_0 \cdots q_k q_{k+1}, u$ et (q_k, a, q_{k+1}) est une transition shift de l'automate. Notez que cette transition empile un état sur la pile.

Il y a une transition de c à c', notée $c \xrightarrow{n \to a_1 \cdots a_k} c'$ si $c = q_0 \cdots q_{i+k}, u$, $c' = q_0 \cdots q_i, nu$, tels que $(q_{i+k}, a_1 \cdots a_k, n)$ est une transition reduce de l'automate et que pour tout $1 \leq j \leq k$, $\lambda(q_{i+j}) = a_j$. Notez que cette transition permet de dépiler des états de la pile : elle dépile les états $q_{i+1}, \cdots, q_k$, elle réduit donc la taille de la pile de k (c'est-à-dire au moins 0, puisqu'on n'a jamais dit que le mot $a_1 \cdots a_k$ était non-vide). Notez également que cette transition ne lit pas de lettre d'entrée (c'est une ε-transition au sens automates). Par contre, elle remet une lettre devant le mot restant à lire. En fait, l'action suivante sera nécessairement un shift de cette lettre, qui dépend évidemment de l'état se trouvant alors en sommet de pile [2]. Dans toute la suite, on considèrera donc les transitions reduce définies ainsi : $c \xrightarrow{\text{red}(n \to a_1 \cdots a_k)} c''$ si il existe c' tel que $c \xrightarrow{n \to a_1 \cdots a_k} c' \xrightarrow{\text{shift}(n)} c''$.

Enfin point important : écrit comme ça, il semble qu'il faille connaître les $k - i$ éléments en sommet de pile pour déterminer si on peut prendre une transition reduce. En pratique, dans ceux que produiront les algorithmes décrits ensuite, cela sera déterminable uniquement via le sommet de pile.

Un calcul π de l'automate est une séquence de configurations et de transitions shift et reduce $c_0, t_0, c_1, \cdots, c_{n-1}, t_{n-1}, c_n$ telle que $c_0 = q_i$, et que pour tout i, c_i et c_{i+1} sont reliés par la transition t_i. Le mot d'entrée $u(\pi)$ lu par le calcul est le mot lu par les transitions shift (en effet, notez que les shift de non-terminaux n'ont lieu qu'au sein des transitions reduce). Les actions reduce du calcul $\text{red}(\pi)$ est la séquence obtenue en ne conservant que les transitions reduce du calcul. Cet objet permettra de reconstruire la dérivation de la grammaire.

Un calcul est acceptant si de plus $c_n = (q_i q_f, \varepsilon)$.

2. La raison en est que c'est la suite de ces deux règles qui correspond à la réduction d'une règle de la grammaire (ce qu'on cherche à faire ici), et que donc c'est assez naturel. C'est d'ailleurs bien comme cela que ce sera défini dans les générateurs de parseurs, et en particulier menhir.

3.3.6 LR0

L'algorithme LR0 a été inventé par Donald Knuth en 1965 [3] et est un algorithme qui, contrairement aux précédents discuté n'est pas capable de parser toutes les grammaires, mais a une bien meilleure complexité que ceux-ci. LR signifie Left-to-right Rightmost derivation, pour signifier que la chaîne de caractères lue le sera exclusivement de gauche à droite (pas de retour en arrière), et que la dérivation de la grammaire ainsi reconnue sera celle où on dérive en premier le non-terminal le plus à droite du mot.

Concrètement, l'algorithme va générer un automate à pile shift-reduce à partir de la grammaire, et ensuite l'exécutera sur les chaînes de caractères fournis en entrée.

Il est assez facile de voir que l'automate lit bien la chaîne de gauche à droite, d'où le Left-to-right. Les actions de réductions correspondront aux règles de la grammaire, et étant donné un calcul π, le miroir de $\mathrm{red}(\pi)$ sera une dérivation droite de la grammaire.

Commençons par définir les états. Intuitivement, l'automate va chercher à déterminer après avoir lu un mot u, quelles sont les règles qui peuvent potentiellement s'appliquer à un mot commençant par u. Pour cela, étant donné une règle $r = N \to \alpha_0 \cdots \alpha_{|r|-1}$, on définit pour chaque entier i entre 0 et $|r|$, l'objet lr0 $r_i = N \to \alpha_0 \cdots \alpha_{i-1} \bullet \alpha_i \cdots \alpha_{|r|-1}$. $\bullet$ est un symbole spécial n'appartenant pas à la grammaire. Cet objet lr0 représente la position de la lecture du mot à droite de la règle correspondant au mot u. On appelle $lr0(r) = \{r_i \mid 0 \leq i \leq |r|\}$ l'ensemble des objets lr0 associés à une règle r.

Si la règle r est $N \to \varepsilon$, alors $lr0(r) = \{N \to \bullet\}$. Ce n'est théoriquement pas un problème, mais la présence d'une telle règle produira un automate avec des conflits.

L'ensemble des états du parseur LR0 associé à une grammaire $G =< \mathcal{N}, \mathcal{T}, S, R >$ est $Q_G = \mathcal{P}(\bigcup_{r \in R} lr0(r))$, i.e., l'ensemble des sous-ensembles des objets lr0 de la grammaire. Cet ensemble a évidemment une taille exponentielle en la taille de la grammaire. En pratique, on se restreindra aux états accessibles, qui dans la plupart des cas d'utilisation ne seront pas trop gros (i.e., polynomiaux).

La clôture d'un objet lr0 $X \to \alpha \bullet Y\beta$ est l'ensemble $\{r_0 = Y \to \bullet\gamma \mid r = Y \to \gamma \in R\}$, à savoir l'ensemble des règles qui dérivent de Y, le non-terminal que l'on peut lire. L'idée étant que si l'on peut lire Y, alors on doit pouvoir lire tout mot engendré par lui.

On ne considèrera que des états clots, c'est-à-dire égaux à leur clôture. Une seule application de la clôture peut ne pas suffire, on devra donc considérer autant d'application que nécessaire jusqu'à stabilisation (ce qui s'obtient en un nombre fini d'étape pour les mêmes raisons que les calculs de Null, First, et Follow).

Étant donné un état lr0 q, on note $\mathrm{clot}(q)$ l'état obtenu en itérant la clôture sur q jusqu'à stabilisation.

L'étiquette d'un état q est fixée par le symbole apparaissant à gauche de $\bullet$ dans l'un des objets LR0 : si q contient un objet LR0 de la forme $X \to \alpha a \bullet \beta$, alors $\lambda(q) = a$. Cela peut sembler mal défini, mais il sera clair dans la suite que tous les états accessible ne peuvent contenir qu'une seule lettre à gauche de $\bullet$ (à cause de la définition des transitions shift). L'état initial ne contient que des règles où $\bullet$ est en première position (raison pour laquelle il n'est pas étiqueté), et tous les autres contiennent soient des règles où $\bullet$ est en première position (introduites par la clôture), soit des règle où la seule lettre à gauche de $\bullet$ est la même.

Étant donné un objet lr0 $l = X \to \alpha \bullet c\beta$ avec $c \in \mathcal{N} \cup \mathcal{T}$, on définit $\mathrm{sh}(l, c) = \{X \to \alpha c \bullet \beta\}$, et $\mathrm{sh}(l, d) = \emptyset$ pour $d \neq c$.

Les transitions shift de l'automate LR0 sont définies par pour tout état LR0 q et lettre $c \in \mathcal{N} \cup \mathcal{T}$, on a $(q, c, \mathrm{clot}(\bigcup_{l \in q} \mathrm{sh}(l, c)))$. Intuitivement, ces transitions correspondent à simplement lire la lettre c dans tous les objets lr0 la contenant comme prochaine lettre. Les objets lr0 ne pouvant pas lire la lettre sont oubliés (ils ne correspondent pas à ce qu'on

est en train de lire). Les transitions qui aboutiraient dans l'état $\emptyset$ ne sont pas présentes dans l'automates (c'est un état d'erreur témoignant que le mot qu'on est en train de lire ne peut être étendu en un mot accepté).

Les transitions reduce sont définies pour les états qui contiennent des règles lues entièrement : si un état q contient un objet LR0 de la forme $X \to \alpha\bullet$, l'automate LR0 contient la transition reduce (q, α, X).

Un point intéressant à remarquer est qu'une transition reduce est toujours applicable depuis une configuration dont le sommet de pile est un état autorisant une transition reduce. En effet, il est assez aisé de prouver via la définition des transitions shift, que si on est dans un état qui contient un objet LR0 $l = X \to \alpha \bullet \beta$, la configuration est contient en sommet de pile des états $q_1 \cdots q_{|\alpha|}$ tels que $\lambda(q_1) \cdots \lambda(q_{|\alpha|}) = \alpha$ (cela peut se prouver par récurrence, en observant que les transitions reduce ne peuvent que préserver cette propriété, puisqu'elles dépilent, puis prennent une transition shift depuis un état de pile où la propriété était vérifiée). Dit autrement, l'état de la pile correspond toujours à un chemin étiqueté par des shift dans l'automate LR0. Ainsi, si on est dans un état qui contient un objet LR0 de la forme $X \to \alpha\bullet$, les $|\alpha|$ états au sommet de la pile sont étiquetés par α, et donc on peut appliquer la transition reduce.

L'idée générale des transitions reduce est que si on a lu entièrement la production d'une règle, alors on la réduit, c'est-à-dire qu'on remplace la production de la règle par le non-terminal qui l'a généré, et on reprend l'analyse après avoir lu ce non-terminal à la place du mot initialement lu.

Il y a une convention particulière pour l'état final qui demande d'avoir un symbole spécial en fin de chaîne. On considère un symbole (terminal) spécial # qui n'est pas dans la grammaire, et on ajoute à la grammaire une règle initiale $S' \to S\#$, où S' est un non-terminal n'appartenant pas à la grammaire, S est le symbole initial. S' est le nouveau symbole initial.

L'état initial est donc $\text{clot}(\{S' \to \bullet S\#\})$, et l'état acceptant est $q_f = \{S' \to S \bullet \#\}$ (on considère que # n'est jamais réellement lu).

En résumé, l'automate LR0 LR0_G d'une grammaire G est le tuple $< Q_G, \mathcal{T}, q_i = \text{clot}(\{S' \to \bullet S\#\}), q_f = \{S' \to S \bullet \#\}, \lambda, \Delta >$ avec

— Q_G l'ensemble des états LR0 de G accessible depuis q_i.
— $(q, a, \text{clot}(\bigcup_{l \in q} \text{sh}(l, c)))$ comme transition shift pour tout état q et toute lettre a.
— λ est définie par la seule lettre apparaissant à gauche de $\bullet$ dans chaque état.
— les transitions reduce définies ci-dessus.

Cet automate peut être défini pour toute grammaire. Néanmoins, il n'est pas utilisable pour toute grammaire : il ne décrit un algorithme de parsing que s'il est déterministe. Ce qui nous amène à la notion de conflit, qui est une manière de caractériser ce non-déterminisme. Il y a deux types de conflits :

— Un état q a un conflit shift-reduce s'il contient à la fois des transitions shift et des transitions reduce.
— Un état q a un conflit reduce-reduce s'il contient au moins deux transitions reduce.

Notez qu'il ne peut y avoir de conflit shift-shift, puisqu'on ne construit qu'une seule transition shift par lettre pour chaque état.

Les deux conflits n'ont pas la même gravité en pratique : les conflits reduce-reduce traduisent d'une ambiguïté de règle de réduction à choisir, et en pratique il n'existe pas de bonne manière de trancher automatiquement ces conflits. Par contre, les conflits shift-reduce traduisent plutôt un ordre de priorité de l'associativité des réductions. Il est possible en pratique d'indiquer à un générateur de parseur comment traiter automatiquement ces cas. Cependant, cela signifie tout de même que l'automate produit est non-déterministe.

Si une grammaire a un automate LR0 sans conflit alors on dit que la grammaire est LR0.

Exemple : $a^n b^n$

On commence par donner un exemple simple d'un langage algébrique non-régulier canonique : $a^n b^n$.

La grammaire usuelle pour cet exemple est constituée d'un seul terminal, et de deux règles $S \to aSb \mid \varepsilon$.

On va modifier cette grammaire pour qu'elle soit plus simple à manipuler dans un cadre LR0, et pour qu'elle corresponde à ce que fait menhir. La première modification consiste à ne pas mettre de ε : si en pratique il est possible d'avoir des règles produisant ε (et on en aura dans le parseur du langage), pour un premier exemple cela serait peu praticable, et de telles grammaires ne sont jamais acceptées par l'algorithme LR0. La seconde va consister à terminer la lecture par le caractère `EOF`, et à ajouter le non-terminal spécial et le terminal spécial considéré dans l'algorithme LR0.

Au final notre grammaire est constitué de trois non-terminaux : `M`, `M'` et `S`, le terminal initial est `M'`, et les règles sont les suivantes :

$$\begin{aligned} M' &\to M\ \# \\ M &\to \texttt{EOF} \\ &\mid S\ \texttt{EOF} \\ S &\to aSb \\ &\mid ab \end{aligned}$$

En TD, vous manipulerez cette même grammaire avec menhir avec le fichier reproduit en suivant (où `M` s'appelle main pour satisfaire au fonctionnement de l'outil) :

```
%token A
%token B
%token EOF

%start <unit> main

%%

main:
| S EOF   {}
| EOF     {}

S:
| A S B   {}
| A B     {}
```

Notez que $main'$ n'y apparaît pas : il est ajouté par menhir automatiquement. On ne le rajoute ici que pour correspondre à ce qui vous observerez en TD.

Si on applique l'algorithme LR0, on obtient l'automate LR0 présenté Figure 3.1. Les règles ajoutées par la clôture sont affichées en bleu. Les règles prêtes à réduire le sont en rouge, et l'état en violet est l'état final. Les règles reduces ne sont pas représentée (par convention, on ne les représente jamais, puisqu'elles dépendent en réalité de la pile : quand

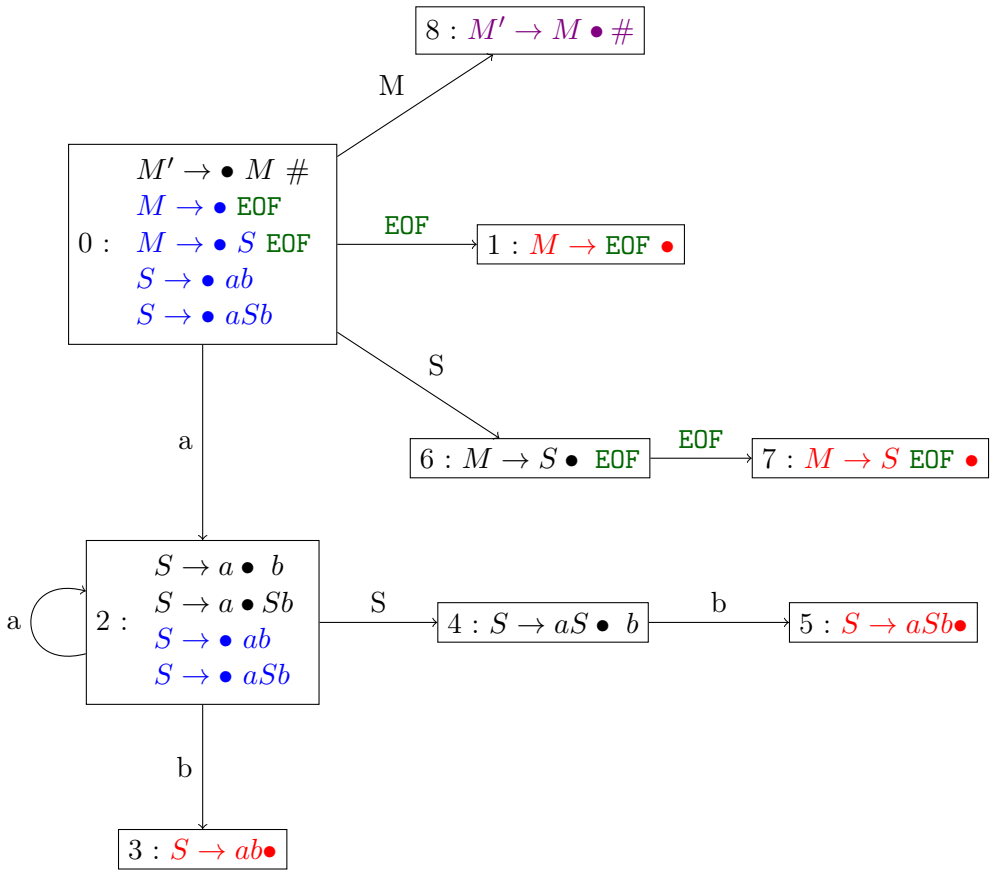


FIGURE 3.1 – L'automate LR0 de la grammaire acceptant $a^n b^n$

on appliquera depuis l'état 3, on pourra se retrouver en fonction de la pile, soit dans l'état 4, soit dans l'état 6).

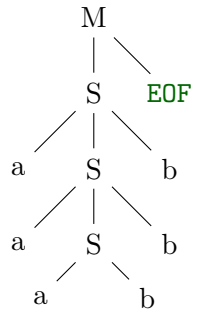
L'automate ainsi produit n'a aucun conflit : aucun état ne contient à la fois de transition shift et de transition reduce. La grammaire qu'on a fournie est donc LR0, et l'algorithme LR0 produira bien un arbre de dérivation.

En TD, vous pourrez générer cet automate avec menhir et l'afficher (avec dot).

Si on considère le mot *aaabbb*, voici l'exécution produite, en affichant la pile sous forme d'une liste avec le sommet à droite :

$$
\begin{aligned}
& [0], aaabbb\texttt{EOF}\# \\
\xrightarrow{\text{shift}(a)} & [0;2], aabbb\texttt{EOF}\# \\
\xrightarrow{\text{shift}(a)} & [0;2;2], abbb\texttt{EOF}\# \\
\xrightarrow{\text{shift}(a)} & [0;2;2;2], bbb\texttt{EOF}\# \\
\xrightarrow{\text{shift}(b)} & [0;2;2;2;3], bb\texttt{EOF}\# \\
\xrightarrow{\text{red}(S\to ab)} & [0;2;2;4], bb\texttt{EOF}\# \\
\xrightarrow{\text{shift}(b)} & [0;2;2;4;5], b\texttt{EOF}\# \\
\xrightarrow{\text{red}(S\to aSb)} & [0;2;4], b\texttt{EOF}\# \\
\xrightarrow{\text{shift}(b)} & [0;2;4;5], \texttt{EOF}\# \\
\xrightarrow{\text{red}(S\to aSb)} & [0;6], \texttt{EOF}\# \\
\xrightarrow{\text{shift}(\texttt{EOF})} & [0;6;7], \# \\
\xrightarrow{\text{red}(M\to S\ \texttt{EOF})} & [0;8], \# \\
& \textit{accept}
\end{aligned}
$$

La dérivation, obtenue en lisant de droite à gauche les transitions de réductions donne l'arbre suivant (on ne place pas le # qui est un symbole technique de menhir) :



Exemple : langage de Dyck

Le langage de Dyck (sur une parenthèse) est le langage des parenthèses bien formées, à savoir l'ensemble des mots qui ont autant de a que de b et pour tout préfixe, plus de a que de b.

Le TD explorera 3 variantes de grammaires générant le langage de Dyck, la grammaire naturelle, qui n'est ni LR0 ni LR1, une variante produisant les arbres de dérivations correspondant à l'associativité à gauche qui est LR0, et une autre produisant les arbres de dérivations correspondant à l'associativité à droite qui n'est pas LR0, mais qui est LR1. On va présenter cette dernière pour faire un lien avec la section suivante sur LR1.

La grammaire, au format menhir, est la suivante :

```
%start <unit> main

%%
```

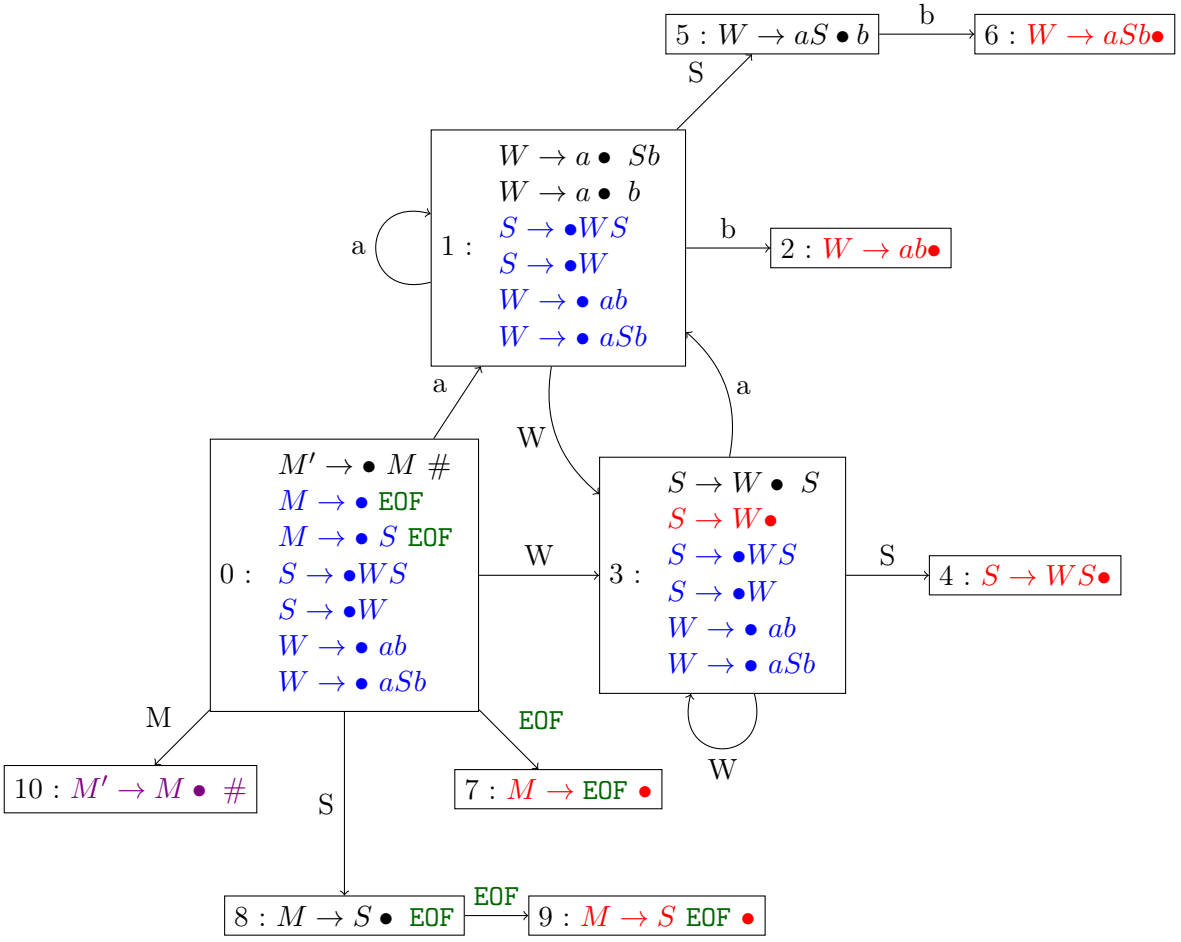


FIGURE 3.2 – L'automate LR0 correspondant à une grammaire de Dyck

```
main:
| seq EOF {}
| EOF     {}

seq:
| word seq {}
| word     {}

word:
| A seq B {}
| A B     {}
```

Par simplicité, on notera main par M, seq par S et word par W.

Si on lui applique l'algorithme LR0, on obtient l'automate de la Figure 3.2.

Cet automate a un conflit shift-reduce à l'état 3. En effet, depuis cet état, il est possible de prendre une transition shift sur la lettre a, et également de réduire la règle $S \to W$. Cela arrive en parsant le mot *abab*, lors de la lecture du second a : on se trouve alors dans l'état $[0; 3]$.

La grammaire n'est donc pas LR0.

En réalité, une rapide observation de l'automate montre que si on réduit alors que la lettre à lire est autre chose que `EOF`, l'automate produira une erreur à l'étape suivante. Cela laisse entendre qu'on pourrait faire mieux sur ce même automate pour détecter ce genre de conflit. Cette observation a historiquement mené à un autre algorithme que nous présentons ensuite.

3.3.7 Simple LR

L'idée de SLR se base sur le problème apparaîssant dans la grammaire précédente : certains conflits shift-reduce de l'automate LR0 sont artificiels, notamment si la lettre suivante ne peut pas être lue si on choisit la transition reduce, alors qu'elle aurait pu être lue par une transition shift.

L'idée est donc de modifier la manière d'utiliser l'automate LR0 en ajoutant à la configuration l'information de la lettre suivante à lire, et de n'autoriser la transition reduce que si cette lettre peut apparaître après le non-terminal à gauche de cette règle.

On définit pour cela, pour chaque non-terminal X, un ensemble $\text{Follow}(X)$ qui est l'ensemble des terminaux qui peuvent apparaître après X dans une dérivation depuis le non-terminal initial. On donne plus loin la définition et le calcul formel de cet ensemble, mais commençons par expliquer comment il s'utilise pour obtenir l'algorithme de parsing SLR.

On définit une configuration SLR comme un couple $(q_0 \cdots q_k, u)$ où les q_i sont des états de l'automate LR0 (comme dans LR0) et u est le mot de terminaux qui reste à lire. En pratique, cela s'obtient en invoquant le lexeur avant de décider de la transition à prendre, et en ne regardant en avance qu'un seul non-terminal (c'est suffisant pour ce qu'on va faire avec SLR). Il serait formellement un peu plus pénible de définir proprement les configurations sans mettre le mot en entier, mais il faut retenir que en pratique, contrairement à ce que laisse entendre ce couple, on a besoin de stocker une seule lettre.

La configuration initiale est (q_i, u) si le mot à lire est u.

Si on est dans une configuration $(q_0 \cdots q_k, au)$, alors on ne peut appliquer une transition reduce de la règle $X \to \alpha$ présente dans q_k que si $a \in \text{Follow}(X)$. Sinon, la transition reduce n'est pas prenable.

À part ce point, SLR se comporte exactement comme LR0.

En particulier, notez que cela ne règle pas tous les conflit shift-reduce (notamment, dans un état contenant $X \to \alpha\bullet$ et $X \to X \bullet a$, il y aura toujours un conflit shift-reduce sur a). Il ne règle aucun conflit reduce-reduce.

Si on reprend l'exemple de la section 3.3.6, il est possible de calculer (cf ci-dessous) de calculer que $\text{Follow}(S) = \{b, \texttt{EOF}\}$. Grâce à cela, le conflit shift-reduce de l'état 3 disparaît : en effet, il n'apparaît que sur la lettre a (puisqu'il n'y a pas de transition shift avec b ou `EOF` depuis 3). Comme $a \notin \text{Follow}(S)$, si on est dans une configuration $(q_i \cdots q_k 3, au)$, alors on prendra la transition shift vers l'état 1, et si on est dans un état $(q_i \cdots q_k 3, bu)$, on prendra la transition reduce de la règle $S \to W$.

On peut donc dire que la grammaire de Dyck donnée est SLR, puisque l'algorithme SLR ne contient pas de conflit.

Calcul de Null, First et Follow

Pour compléter cette section sur SLR, il faut donner formellement la méthode de calcul de Follow. Pour cela, on a besoin de deux autres fonctions sur les éléments de la grammaire, Null et First, qui avec Follow devraient vous rappeler l'algorithme de Glushkov sur les automates finis (puisqu'ils ont essentiellement le même sens).

Null détermine si un mot permet, ou non, de dériver ε :

Définition 3.3.3 (Null)**.** *Soit* $\alpha \in (\mathcal{N} \uplus \mathcal{T})^*$, $\mathrm{Null}(\alpha) = \top$ *si et seulement si* $\alpha \xrightarrow[\mathcal{G}]{*} \varepsilon$.

On peut le calculer récursivement, mais en faisant attention à un point : le calcul récursif naïf ne termine pas ! Pour régler cela, il suffit d'observer que si on peut dériver ε depuis un non-terminal, alors on peut le faire avec un arbre sans répétition sur une branche. On regarde donc plutôt $\mathrm{Null}(\alpha, \mathcal{X})$, avec $\mathcal{X} \subseteq \mathcal{N}$.

Il se calcule récursivement comme suit :

$$\begin{aligned}
\mathrm{Null}(a, \mathcal{X}) =& \bot && \text{pour } a \in \mathcal{T} \\
\mathrm{Null}(\varepsilon, \mathcal{X}) =& \top \\
\mathrm{Null}(X, \mathcal{X}) =& \bigvee_{X \to \alpha} \mathrm{Null}(\alpha, \mathcal{X} \backslash \{X\}) && \text{si } X \in \mathcal{X} \\
\mathrm{Null}(X, \mathcal{X}) =& \bot && \text{si } X \notin \mathcal{X} \\
\mathrm{Null}(\alpha_1\alpha_2, \mathcal{X}) =& \mathrm{Null}(\alpha_1, \mathcal{X}) \wedge \mathrm{Null}(\alpha_2, \mathcal{X}).
\end{aligned}$$

On a au final pour tout α, $\mathrm{Null}(\alpha) = \mathrm{Null}(\alpha, \mathcal{N})$.

First désigne les terminaux pouvant apparaître en première position des mots dérivés depuis un mot :

Définition 3.3.4 (First)**.** *Soit* $\alpha \in (\mathcal{N} \uplus \mathcal{T})^*$, $\mathrm{First}(\alpha) = \{a \in \mathcal{T} \mid \alpha \xrightarrow[\mathcal{G}]{*} av\}$.

Pour le calculer, il faut faire la même observation que pour Null, à savoir qu'on peut avoir la réponse en ne considérant que les dérivations sans répétition sur une branche. On définit donc $\mathrm{First}(\alpha, \mathcal{X})$:

$$\begin{aligned}
\mathrm{First}(\varepsilon, \mathcal{X}) =& \emptyset \\
\mathrm{First}(a, \mathcal{X}) =& \{a\} && \text{si } a \in \mathcal{T} \\
\mathrm{First}(X, \mathcal{X}) =& \emptyset && \text{si } X \notin \mathcal{X} \\
\mathrm{First}(X, \mathcal{X}) =& \bigcup_{X \to \alpha} \mathrm{First}(\alpha, \mathcal{X} \backslash \{X\}) && \text{si } X \in \mathcal{X} \\
\mathrm{First}(\alpha_1\alpha_2, \mathcal{X}) =& \mathrm{First}(\alpha_1, \mathcal{X}) && \text{si } \mathrm{Null}(\alpha_1) = \bot \\
\mathrm{First}(\alpha_1\alpha_2, \mathcal{X}) =& \mathrm{First}(\alpha_1, \mathcal{X}) \cup \mathrm{First}(\alpha_2, \mathcal{X}) && \text{si } \mathrm{Null}(\alpha_1) = \top
\end{aligned}$$

On a finalement $\mathrm{First}(\alpha) = \mathrm{First}(\alpha, \mathcal{N})$.

Follow désigne l'ensemble des terminaux pouvant apparaître comme première lettre d'un mot apparaissant après un non-terminal :

Définition 3.3.5 (Follow)**.** *Soit* $X \in \mathcal{N}$, $\mathrm{Follow}(X) = \{a \in \mathcal{T} \mid S \xrightarrow[\mathcal{G}]{*} uXav\}$.

De la même manière, que les deux précédents, une définition naïve sera cyclique, mais on s'en sort en remarquant que le résultat peut être obtenu en ne considérant que des arbres sans répétitions (même si ici le calcul remonte dans l'arbre plutôt que de descendre). On considère donc $\mathrm{Follow}(X, \mathcal{X})$:

$$\begin{aligned}
\mathrm{Follow}(X, \mathcal{X}) =& \bigcup_{Y \to \alpha X \beta} \mathrm{First}(\beta) \cup \bigcup_{Y \to \alpha X \beta \mid \mathrm{Null}(\beta) = \top} \mathrm{Follow}(Y, \mathcal{X} \backslash \{X\}) && \text{si } x \in \mathcal{X} \\
\mathrm{Follow}(X, \mathcal{X}) =& \emptyset && \text{si } X \notin \mathcal{X}
\end{aligned}$$

On a $\text{Follow}(X) = \text{Follow}(X, \mathcal{N})$.

Note : en réalité, les ensembles ci-dessus sont plus efficacement calculés par point fixe, ce qui consiste à itérer des calculs jusqu'à stabilisation (en gros les mêmes). Pour simplifier la compréhension, on préfère donner cette version moins efficace, mais requérant moins d'abstraction et équivalente.

3.3.8 LR1

En pratique, la classe des grammaires LR0 est très restrictive et la plupart des grammaires naturelles pour les langages de programmations n'y entrent pas. Il existe des extensions de cette analyse qui reposent principalement sur le fait de retenir plus d'information dans les états, à savoir les terminaux qui peuvent apparaître après la règle que l'on est en train de lire. SLR est un premier pas dans cette direction. Cette approche a eu du succès puisqu'elles permettent de garder un parseur relativement économe en espace (de l'ordre de celui LR0) tout en étant plus fine.

Cependant, elle garde des conflits assez artificiels (en particulier reduce-reduce puisque ceux-ci sont les mêmes que LR0) pour des grammaires pourtant naturelles. Il y a donc une idée de modifier l'automate pour avoir des états qui dépendent des lettres que l'on va lire (comme pour SLR mais en ayant des informations qui en dépendent dans l'automate et pas seulement sa sémantique).

Cela donne lieu à la classe des algorithmes LRk, où k est le nombre de lettres que l'on retient. En pratique LR1 est suffisant, et comme c'est ce que menhir implémente, c'est celui dont nous allons discuter ici. Historiquement il a également été introduit par Donald Knuth, mais avec une taille d'automates prohibitive, qui a longtemps empêché son implémentation en pratique. Menhir se base sur un autre algorithme de génération d'automate LR1, l'algorithme de Pager[4], qui génère un automate de taille la plupart du temps comparable à l'automate LR0 (mais pouvant être un peu plus gros).

La plupart des outils existant (les plus anciens comme yacc, mais également des plus récents) utilisent en fait une approximation de LR1, nommée LALR qui consiste à fusionner de force tous les états ayant le même noyau LR0 (et de ce fait de forcer à ce que la taille soit exactement celle de l'automate LR0). De ce fait, on réintroduit des conflits qui peuvent être artificiels.

Il peut paraître surprenant que des outils pourtant plus récent que l'algorithme de Pager soient restés sur LALR, mais outre la difficulté théorique plus grande de cet algorithme, c'est également dû au fait que la plupart des grammaires utiles pour un langage de programmation sont modifiables assez simplement pour être LALR (au point que trouver une grammaire LR1 non LALR relativement naturelle est tellement peu simple, que je n'en ai pas trouvée). La question paralèle est donc pourquoi menhir s'embête avec cet algorithme moins simple à implémenter ? La raison en est que comme dit plus haut, dans LALR, on réintroduit des conflits plus artificiels, et c'est lorsque la grammaire considérée n'est pas LR1 que la différence apparaît : les erreurs dans ce cas seront plus dures à comprendre et le rapport d'erreur d'un générateur de parseur comme yacc ou bison est en pratique moins agréable et lisible que celui de menhir.

On peut maintenant présenter l'automate LR1. Cet automate est également un automate à pile shift-reduce, très similaire à l'automate LR0, mais où les objets apparaissant dans les états sont plus précis (et plus nombreux).

Un objet lr1 est un couple (l, a) où l est un objet lr0 et a un terminal. Le sens d'un tel objet sera, en plus de celui de l'objet lr0 qui décrit où en est la lecture de la règle décrite par l, de dire qu'après la lecture de cette règle, il est possible de lire la lettre a. L'idée

derrière cet objet est de désambiguiser les conflits shift-reduce en ne réduisant une règle $r = X \to \alpha$ que si l'état courant contient $(X \to \alpha\bullet, a)$ et que la prochaine lettre à lire est un a. Dans le cas contraire, on prendra la transition shift.

On définit la clôture d'un objet lr1 : $\text{clot}_{lr1}(X \to \alpha \bullet Y\beta, a) = \{(Y \to \bullet\gamma, b) \mid r = Y \to \gamma \in R, b \in \text{First}(\beta a)\}$.

Les états de l'automate LR1 sont les ensembles d'objets lr1 clos.

Les transitions shifts sont définies comme dans le cas lr0 :

$$\text{sh}_{lr1}((X \to \alpha \bullet c\beta, a), d) = \begin{cases} \{(X \to \alpha c \bullet \beta, a)\} \text{ si } c = d \\ \emptyset \text{ sinon} \end{cases}$$

Et on a donc pour tout état q et lettre c, la transition shift $(q, c, \text{clot}_{lr1}(\bigcup_{(l,a)\in q} \text{sh}_{lr1}((l, a), c)))$.

Les transitions reduce sont aussi définies comme dans le cas LR0.

La différence n'a pas l'air évidente, mais ce qui va changer, c'est l'action à prendre lorsqu'on est dans un état et surtout la notion de conflits.

Si on est dans un état q et que la prochaine lettre à lire est un a, alors :

— On peut appliquer une transition shift s'il y a une définie sur a (comme pour LR0).

— On peut appliquer la transition reduce correspondant à $l = X \to \alpha\bullet$ si et seulement si l'état contient (l, a).

La notion de conflit est donc un peu plus subtile :

— Un état q a un conflit shift-reduce s'il contient à la fois une transition shift pour une lettre a et contient un objet lr1 $(l = X \to \alpha\bullet, a)$.

— Un état q a un conflit reduce-reduce s'il contient deux objets lr1 $(l = X \to \alpha\bullet, a)$ et $(l' = Y \to \beta\bullet, a)$ pour $l \neq l'$

En particulier, le lookahead permet donc de désambiguiser des conflits qui apparaissent dans un automate LR0 mais peuvent être évités grâce à la précision supplémentaire apportée par LR1. Et contrairement à SLR, ici certains conflits reduce-reduce de l'automate LR0 ne sont pas présent (si les ensembles de lettres à lire sont disjoints, ce n'est pas un conflit).

Il y a des grammaires pour lesquelles l'automate LR1 n'a pas de conflits alors que l'automate LR0 en a, et donc cet algorithme permet en pratique de construire des parseurs pour plus de grammaires. Cela permet essentiellement une meilleure facilité d'utilisation, et la plupart des langages de programmation peuvent être mis en forme LR1.

Cependant, l'automate engendré par la construction naïve d'un automate LR1 (en générant les états itérativement) qu'on appelle automate LR1 canonique aboutit en pratique à des automates très gros, qui ne peuvent être implémentés en pratique. C'était évidemment encore plus vrai à l'époque où ces algorithmes ont été conçus (fin des années 60).

L'outil que nous utiliserons, menhir, produit lui un analyseur LR1 complet. Pour ce faire, il ne construit pas l'automate LR1 canonique, mais implémente un algorithme conçu par Pager en 1977[4] qui fusionne les états LR1 lors de la construction à la volée lorsque c'est possible.

Cet algorithme consiste à fusionner les états lr1 ayant le même noyau lr0 dans des conditions particulières (qu'on ne détaillera pas ici) qui assure que la fusion des deux états ne peut créer de conflit, et que l'automate ainsi produit est équivalent à l'automate LR1. En pratique, la taille de l'automate produit est en général équivalente à l'automate LR0 (quelques états sont dupliqués, mais assez peu).

En comparaison, la technique LALR se base sur la même idée, mais elle fusionne systématiquement les états de même noyau pour produire un automate avec les mêmes états que l'automate LR0, mais avec des informations sur le lookahead plus fine que SLR. En revanche, ce faisant des conflits peuvent apparaître.

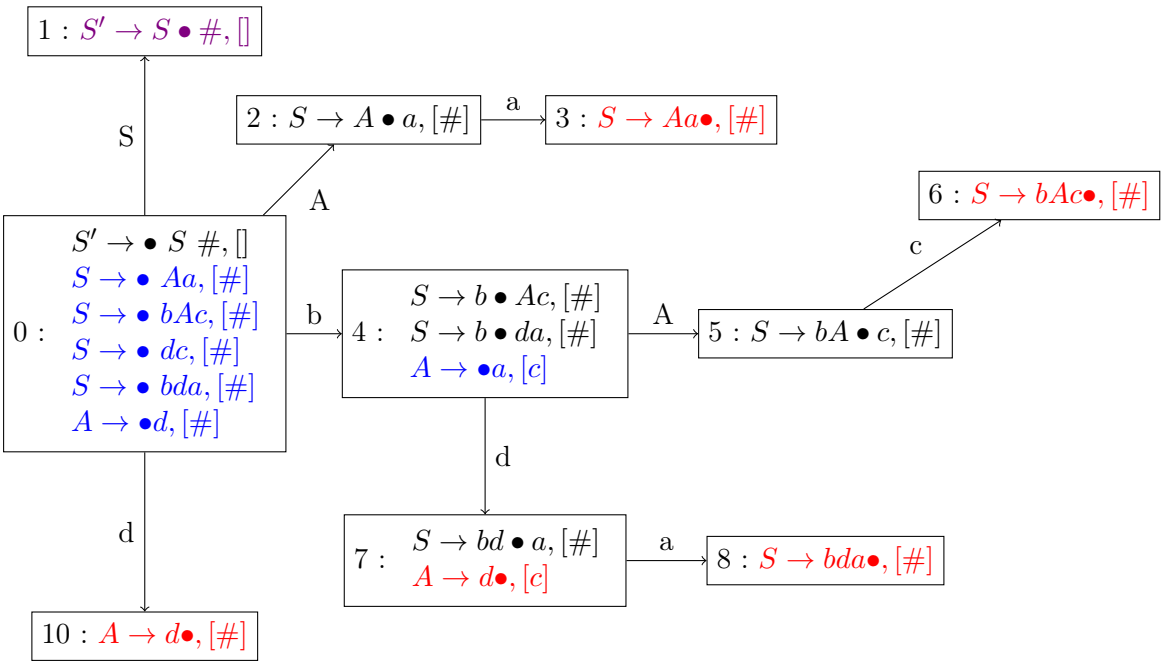


FIGURE 3.3 – L'automate LR1 canonique de la grammaire exemple

Menhir offre donc une meilleur capacité d'expression de grammaire à l'utilisateur que ceux basés sur LALR, bien que ce soit à la marge.

3.3.9 Exemple

La grammaire suivante est LR1, mais pas SLR :

```
S :
| A a
| b A c
| d c
| b d a

A :
| d
```

Si on construit l'automate LR0 (figure 3.3 en omettant les informations de lookahead), on observe que l'état atteint (7) en lisant bd contient à la fois $S \to bd \bullet a$ et $A \to d\bullet$.

Il y a donc un conflit shift-reduce dans cet état dans l'algorithme LR0, puisqu'on peut lire a.

Si on calcule Follow(A), on trouve que cela vaut $\{a, c\}$, et donc le conflit subsiste avec l'algorithme SLR, puisqu'on peut lire a, et que a peut suivre A.

Cependant, si on applique la construction LR1 (figure 3.3), alors l'état obtenu (7) contient les objets $(S \to bd \bullet a, \#)$ et $(A \to d\bullet, c)$. Il n'y a donc plus de conflit : si la lettre suivante est un a, on ne peut pas réduire A, puisque dans ce contexte, a ne peut suivre A, donc on prend le shift. On ne réduira A que si la lettre suivante est un c.

3.3.10 Priorités et conflits dans menhir

Maintenant qu'on a vu en théorie ce qu'est un conflit et d'où il vient, on peut voir comment régler cela avec menhir (et d'autres outils).

La grammaire présentée en section 3.3.2 est ambiguë. En effet `1 + 2 + 3` peut être obtenue soit comme `(1 + 2)+ 3` soit comme `1 + (2 + 3)`. Ici, cette ambiguïté grammaticale n'est pas sémantique, mais seulement syntaxique donc une résolution arbitraire conviendra, mais si on ajoute d'autres opérations (la multiplication, par exemple), cela devient important.

Il y a deux manière de s'en sortir : modifier la grammaire – on verra en TD comment le faire pour le cas arithmétique – ce qui est fastidieux et peut produire des grammaires peu naturelles, où donner des règles de priorité pour régler automatiquement les conflits shift-reduce ou reduce-reduce qui surviennent. Pour ce second point, il est possible d'associer une associativité (gauche ou droite), ainsi que des priorité à des tokens.

Les associativités peuvent être à gauche et à droite. Elles se notent par `%left term` ou `%right term`, pour `term` un terminal. Si un terminal a une associativité gauche, alors lors d'un conflit shift-reduce concernant ce terminal, on appliquera le reduce, si l'associativité est droite, on appliquera le shift. Cela arrive, dans la grammaire précédente dans les états contenant à la fois $X \to expr \bullet$ `ADD` $expr$, `ADD` et $X \to expr$ `ADD` $expr \bullet$, `ADD`.

L'ordre dans lequel ces déclarations d'associativités sont faites va induire un ordre de priorité (qui est implicite). Si on écrit `%left ADD MUL`, ou `%left ADD` puis `%right MUL` sur la ligne suivante, `ADD` aura une priorité plus faible que `MUL`, ce qui fera que les conflits shift-reduce impliquant ces deux terminaux seront toujours tranchés en faveur de `MUL`. Par exemple, si on est dans un état contenant $X \to expr \bullet$ `ADD` $expr$, `ADD` et $X \to expr$ `MUL` $expr \bullet$, `ADD`, et qu'on lit un `ADD`, c'est la transition reduce qui sera prise. Cela a pour effet que la chaîne $1 + 2 * 3$ sera interprétée comme $1 + (2 * 3)$, ce qui est bien ce que l'on veut.

Il est possible d'ajouter des terminaux dans l'ordre de priorité sans leur associer une associativité (parce qu'ils n'ont pas à en avoir et qu'on veut les erreurs correspondantes) avec le mot-clef `%nonassoc`.

Ces règles permettent de manipuler des grammaires ambiguës dans menhir mais plus naturelles en exprimant naturellement la résolution des conflits qui surviennent. Cela permet une conception plus naturelle de la grammaire, mais demande une compréhension des concepts d'associativité et de conflits.

En pratique, voici comment menhir interprète ces informations. Chaque production hérite par défaut de la priorité de son token le plus à droite. En cas de conflit reduce-reduce, la règle avec la priorité la plus haute est réduite. En cas de conflit shift-reduce, on compare la priorité du token à shifter avec celle de la règle, et on effectue celui dans la priorité est la plus haute. Si le token et la règle ont la même priorité, alors on regarde la règle d'associativité du token : si l'associativité est à gauche, on effectue le reduce, si elle est à droite, on effectue le shift. Si il est non-associatif, le conflit demeure.

Parfois, ce comportement n'est pas celui souhaité, ou on veut désambiguïser les conflits de règles sans terminaux. Dans ce cas, on peut associer une priorité à une règle en lui associant la même priorité qu'un terminal. Cela se fait en ajoutant l'annotation `%prec term` après la production de la règle en question.

Si on reprend la grammaire de la section 3.3.2, on n'aura aucun conflit si on l'écrit comme suit :

```
%token <int> INT
%token ADD
```

```
%left ADD
%start <int> expr

%%

expr:
| i = INT { i }
| v1 = expr ADD v2 = expr { v1 + v2 }
```

3.3.11 Gestion d'erreurs

Un dernier point à discuter est ce qu'il se passe en cas d'erreur. Il y a, dans les outils de parsing plusieurs approches existantes, notamment une approche historique et une approche introduite par menhir.

Historiquement, le parsing d'un fichier de code mettait beaucoup de temps, et si une erreur de syntaxe se glissait, on ne le découvrait que plus tard. Si l'outil avait terminé sur une erreur brusquement, cela laissait peu d'information. Cela a donné lieu à l'élaboration de toute une batterie de techniques pour continuer le calcul en cas d'erreur, en réparant le code à la volée, pour continuer le parsing, et au minimum avoir un rapport d'erreur ne parlant pas uniquement de la première erreur.

Dans des outils tels que yacc, bison, ocamlyacc (ou menhir qui le supporte toujours), l'idée est d'avoir un token spécial, `error` qui ne peut pas apparaître dans le lexeur. Lorsque le parseur se retrouve dans un état où il ne peut prendre aucune transition (ni shift, ni reduce), alors il prend une transition shift avec ce token `error`. Les règles de la grammaire peuvent contenir ce token `error` qui permettent de faire quelque chose de ce token et de reprendre le calcul après la survenue de l'erreur. Dans le cas où il n'y a pas de telle règle reduce applicable, le parseur termine sur une erreur.

Cette technique peut par exemple permettre d'ignorer une erreur dans une instruction (tout en la reportant avec un affichage, par exemple), tout en continuant le parsing (pour détecter des erreurs suivantes). Pour cela, il faudrait par exemple ajouter une règle de la forme `error SEMICOLON {printf "erreur"; BLOCK []}` à la règle traitant le cas instr (un truc du genre, ce que j'écris là n'est absolument pas suffisant). Attention, cela ne fonctionnera pas avec le visualiseur (qui utilise la technique suivante). Par contre, vous pouvez la tester avec interpreter.out, si vous le souhaitez.

Menhir introduit une nouvelle technique de gestion d'erreur, qui à ma connaissance est particulière à cet outil, et qui repose sur des résultats récents[2, 5], notamment de l'auteur de menhir, François Pottier. Nous ne ferons que l'esquisser ici. La technique repose sur le fait que menhir peut être exécuté pas à pas. Lorsqu'une erreur survient, le parseur est mis en pause et rend la main au programme appelant le parseur. Celui-ci peut ensuite réaliser une récupération à la main (en manipulant directement l'état du parseur par exemple), mais surtout permet d'utiliser l'état courant du parseur pour générer un message d'erreur approprié.

Cette nouvelle technique est facilité par la capacité de menhir à générer, à partir d'une grammaire, tous les cas d'erreurs pouvant survenir, et de permettre à l'utilisateur de donner pour chacun un message expliquant le problème rencontré. Ces cas sont décrits par rapport à l'état de l'automate atteint lors de l'erreur, ce qui permet à l'utilisateur de pouvoir expliquer plus finement le problème rencontré. C'est là la principale différence avec la technique «traditionnelle» : dans celle-ci, l'erreur est capturée par la règle contenant `error`, ce qui peut correspondre à plusieurs erreurs auxquelles sont alors associées un seul message.

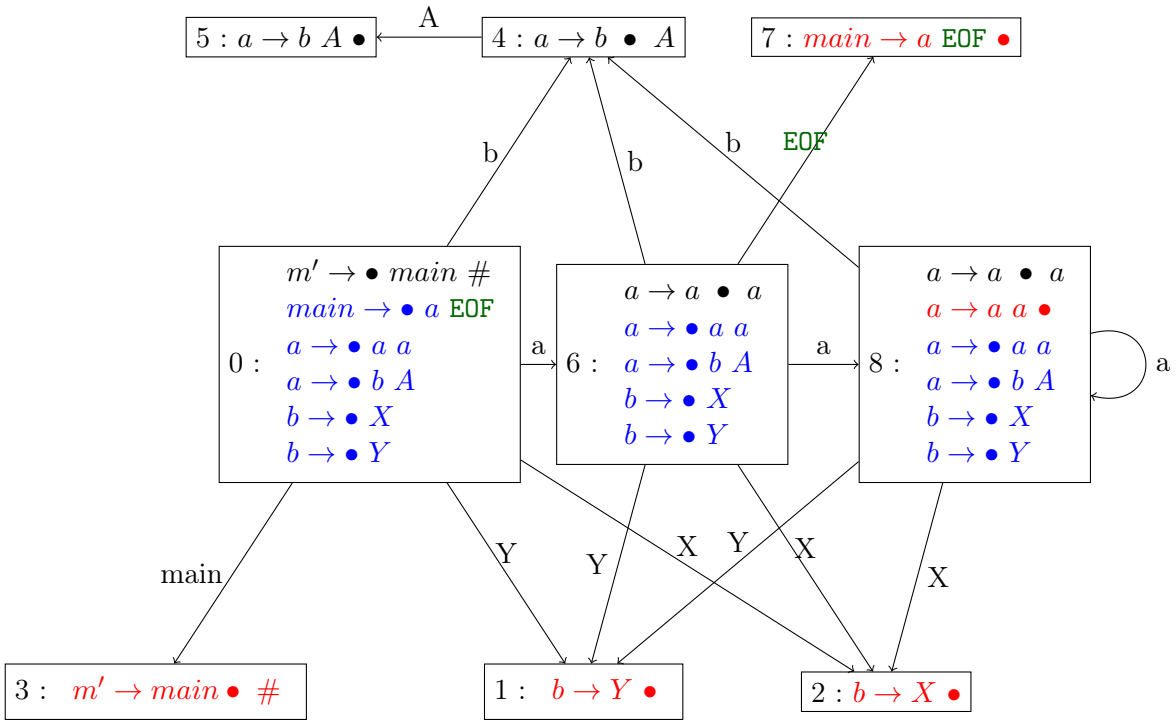


FIGURE 3.4 – L'automate LR0 correspondant à la grammaire de la section 3.4

Cette technique n'est cependant pas si simple à utiliser – elle demande en particulier une bonne compréhension de l'algorithme générant l'automate shift-reduce, et nous ne développerons donc pas plus ce point. Plus de détails peuvent être trouvé dans le manuel de menhir.

3.4 Résumé par un exemple/questions types

On utilise la grammaire suivante comme exemple :

```
%token EOF A X Y
%start <unit> main

%%

main:
| a EOF {}

a:
| a a {}
| b A {}

b:
| X {}
| Y {}
```

• Première question : la grammaire est-elle LR0 ? Cela revient à ce demander si l'automate LR0 de cette grammaire produit des conflits.

Pour y répondre, on construit l'automate LR0.

L'état initial est l'état dont le noyau est constitué de l'objet lr0 correspondant à l'état initial, en position initiale, donc ici, $main \to \bullet a\texttt{EOF}$.

Il faut ensuite calculer la clôture lr0 de cet objet pour avoir l'état LR0 complet. Cette clôture correspond à introduire toutes les règles pouvant être lues après la position courante. Ici, comme à droite du $\bullet$ se trouve un a, on ajoute toutes les règles pouvant être dérivées depuis a. On continue cette clôture jusqu'à ce que l'ensemble obtenu soit stable.

Ici, on obtient l'état constitué des objets lr0 suivants : $main \to \bullet a\texttt{EOF}$, $a \to \bullet aa$, $a \to \bullet bA$, $b \to \bullet X$ et $b \to \bullet Y$.

À partir de cet état, on calcule ses successeurs par chaque shift possible. Pour chaque règle, on fait avancer le $\bullet$ d'une lettre si la lettre correspond au shift que l'on fait. Sinon la règle disparaît. Par exemple, pour l'état initial, en faisant un shift de a, on obtient les deux objets lr0 suivants : $main \to a \bullet \texttt{EOF}$ et $a \to a \bullet a$. Il faut ensuite, à nouveau prendre la clôture de l'ensemble obtenu pour obtenir l'état atteint.

On continue ce processus jusqu'à ce que tous les états aient leurs transition shift sortante.

Chaque état contenant un (ou plusieurs) objets lr0 avec $\bullet$ en dernière position contiennent alors une transition reduce de la règle correspondante.

Dans l'exemple donné ci-dessus, l'automate à obtenir est laissé en exercice.

La grammaire obtenue est LR0 si et seulement si l'automate ainsi obtenu ne contient aucun conflit, c'est-à-dire aucun état ne contenant à la fois des transitions shift (de terminaux) et reduce, ou plusieurs transitions reduce.

Dans cet exemple, on atteint l'état 8 contenant les objets $a \to aa\bullet$, $a \to a \bullet a$, $a \to \bullet aa$, $a \to \bullet bA$, $b \to \bullet X$ et $b \to \bullet Y$. Dans cet état on peut à la fois réduire $a \to aa\bullet$ et shifter X et Y. On a donc un conflit et la grammaire n'est pas LR0.

• Seconde question, cet grammaire est-elle SLR, c'est-à-dire, en regardant une lettre de lookahead et en regardant les ensembles Follow des non-terminaux de gauche des règles impliquées dans des conflits, peut-on désambiguïser les conflits en question ?

On calcule donc Follow(a). Pour cela, on commence par regarder dans quelles règles (et à quelle position) ce non-terminal apparaît. On en déduit que Follow(a) est l'union de $\{\texttt{EOF}\}$ (par la règle $main \to a\texttt{EOF}$), de First(a) (par le premier a de $a \to aa$ qui est suivi par un a) et de Follow(a) (par le second a de $a \to aa$). Le Follow(a) réapparaissant mènerait à une récurrence infinie sans rien apporter, on ne le redéveloppe donc pas et on le remplace par $\emptyset$.

On a donc Follow(a) $= \{\texttt{EOF}\} \cup$ First(a) et il reste donc à calculer ce dernier. Pour cela, on regarde les première lettres des règles dérivées à partir de a. On a First(a) $=$ First(a) $\cup$ First(b). On supprime le First(a) (qui mènerait à une définition circulaire) et on calcule First(b) $= \{X, Y\}$ (par ses deux règles, qui commencent par des terminaux).

Au final, on a Follow(a) $= \{\texttt{EOF}, X, Y\}$.

Le conflit de l'état plus haut n'est donc pas réglé, puisque lorsque la prochaine lettre à lire est un X, on peut tout autant le voir après un a (en prenant la réduction) que le shifter directement.

SLR ne désambiguïse donc pas cet automate, et la grammaire n'est donc pas SLR.

• Troisième question : comment régler ce conflit avec des annotations ?

On regarde quels sont les conflits shift-reduce impliqués. Ici, on voit que les (seuls) conflits impliquent un shift de X ou de Y avec la réduction de $a \to aa$, qui ne contient aucun token. Il faut donc associer une priorité à chacun des éléments suivants : X, Y et un token fictif `FICT` qu'on accole à la règle $a \to aa$ avec l'annotation `%prec FICT`.

En fonction de l'ordre de priorité, le parseur favorisera la règle de priorité supérieure. Ici il n'y a pas de question d'associativité puisqu'un token n'est jamais en conflit avec une règle portant la même priorité que lui.

Il faut considérer les arbres que l'on souhaite accepter lorsqu'on choisit ces priorités, puisque quand de telles annotations sont nécessaires, c'est en général le signe que la grammaire est ambiguë.

Enfin attention, les désambiguïsations de conflits sont uniformes dans l'automate, c'est-à-dire qu'il est impossible de trancher un conflit entre shifter un token X et réduire une règle r en favorisant le shift dans un état q_1 et la réduction dans un état q_2 (ce qui n'est pas étonnant puisqu'on donne la grammaire et non l'automate).

- Dernière question possible : quelle est l'exécution (ou les exécutions) sur un mot donné et à quels arbres cela correspond-t'il ?

Dans cet exemple, on peut considérer le mot $XAYAXA$`EOF`.

Une exécution possible est la suivante (en ne notant que les configurations et les transitions) :

$$(0) \xrightarrow{X} (0,2) \xrightarrow{red(X\to X)} (0,4) \xrightarrow{A} (0,4,5) \xrightarrow{red(a\to bA)} (0,6) \xrightarrow{Y} (0,6,1) \xrightarrow{red(b\to Y)} (0,6,4) \xrightarrow{A} (0,6,4,5) \xrightarrow{red(a\to bA)} (0,6,8) \xrightarrow{red(a\to aa)} (0,6) \xrightarrow{X} (0,6,2) \xrightarrow{red(b\to X)} (0,6,4) \xrightarrow{A} (0,6,4,5) \xrightarrow{red(a\to bA)} (0,6,8) \xrightarrow{red(a\to aa)} (0,6) \xrightarrow{\texttt{EOF}} (0,6,7) \xrightarrow{red(main\to a\texttt{EOF})} (0,3).$$

Cette exécution correspond à avoir privilégier le reduce sur le shift (la réduction de $a \to aa$ depuis (0,6,8)). Elle correspond donc à l'arbre $main(a(a(a(b(X),A),a(b(Y),A)),a(b(X),A)),\texttt{EOF})$.

À finir plus tard (mettre au propre).

Chapitre 4

Analyse sémantique

Avec les sections précédentes, nous avons un interpréteur de notre langage complet, qui sur des programmes corrects en applique la sémantique. Néanmoins, notre interpréteur accepte d'interpréter des programmes qui sont voués à échouer, c'est-à-dire qui n'ont pas de sémantique définie. C'est en particulier le cas si l'on applique des opérations sur des valeurs inconsistantes (additionner un entier et un flottant ou prendre le et de deux entiers, par exemple). Dans notre implémentation de l'interpréteur, le programme renverra une exception, on pourrait donc croire que ce n'est pas extrêmement grave. Cependant, cela donne des informations insuffisantes à l'utilisateur, et lorsque l'on produira du code assembleur, on ne pourra systématiquement renvoyer des exceptions dans ce cas, on sera tributaire de l'effet des instructions choisies, ce qui provoquera un comportement non prévu par la sémantique, et donc une erreur silencieuse. Ce dernier point est le plus dramatique : cela provoquera des programmes avec des comportements fautifs durs à détecter, pouvant causer de graves bugs.

Une solution à ce problème est d'ajouter une phase d'analyse qui va chercher à détecter ces problème avant de commencer l'interprétation ou la production de code assembleur et de reporter ces problèmes à l'utilisateur.

Le but de cette section est de résumer le type de vérification et de transformations que l'on peut faire à ce niveau, ainsi que de discuter des différentes options pour le faire. Tout ne sera pas abordé en travaux pratique (puisque nous aurons à faire des choix à faire pour notre langage).

En particulier, nous nous concentrerons sur la vérification de type et le report d'erreurs à l'utilisateur.

4.1 Comment faire cette analyse

Il existe plusieurs techniques pour faire cette analyse, qui globalement reviennent toutes à explorer l'arbre de syntaxe abstraite pour l'analyser et en déterminer les erreurs.

L'une des techniques qui a émergé dans les débuts des compilateurs consiste à faire cette analyse lors de l'analyse syntaxique. En réalité, cela est dû à ce que dans les premiers compilateurs, on ne produisait pas nécessairement de code intermédiaire en sortie du parseur, mais on en générait directement le code assembleur. L'un des principaux avantages de cette technique est que l'arbre syntaxique n'est jamais produit explicitement et qu'on ne l'explore qu'une seule fois. Sur des machines des années 70, ces considérations de performances étaient essentielles, et donc ce choix était donc courant. C'est d'ailleurs pour cela qu'on parle de «compilateur de compilateurs» pour les générateurs de parseurs, puisqu'on peut y écrire un compilateur entier.

```
%token SEMICOLON ";" ASSIGN "=" EOF
%token <string> ID
%token <int> INT

%start <(string,int) Hashtbl.t -> unit> main

%%
main:
| i = instr_l EOF {fun h -> i h}

instr_l:
| i = instr l = instr_l {fun h -> i h; l h}
| {fun _ -> ()}

instr:
| v = ID "=" i = INT SEMICOLON { fun h -> Hashtbl.replace h v
   i}
| v = ID "=" v2 = ID SEMICOLON { fun h -> Hashtbl.replace h v
   (Hashtbl.find h v2)}
```

FIGURE 4.1 – Un interpréteur entièrement en menhir

Néanmoins, cette technique n'est pas exempte d'inconvénients. En particulier, le code des actions sémantiques devient beaucoup plus long que ce qu'on a pu faire – et donc plus facile à y commettre des bugs. Par ailleurs, l'ordre d'évaluation des branches de l'arbres y devient beaucoup plus crucial, les actions pouvant alors faire des effets de bords, pas toujours simples à prédire.

Cela a donné lieu à la théorie des grammaires attribuées qui prennent en considération ces effets.

Pour ce qui concerne nos outils, cette technique donnerait lieu à des non-terminaux ayant des types fonctionnels et le parseur générerait une fonction au lieu d'un arbre de syntaxe abstraite.

On peut trouver figure 4.1 un parseur écrit en menhir étant l'interpréteur complet d'un mini-langage d'affectations.

Une autre famille de techniques, plus modulaire, consiste à utiliser le parseur uniquement pour générer un arbre de syntaxe abstrait, et d'analyser ensuite celui-ci pour y apposer des annotations et utiliser ces mêmes annotations pour détecter les erreurs. On peut également modifier l'arbre à la volée.

C'est l'approche que l'on choisit dans le cadre de ce cours. OCaml est un langage particulièrement adapté à ce choix, puisque le pattern-matching nous permet de définir aisément des traitement adaptés à chaque partie de l'AST.

L'un des avantages appréciables de cette approche est qu'elle permet de définir une fonction par type d'analyse et donc de les considérer séparément, de manière à avoir un code plus compréhensible (et donc maintenable). Évidemment, le désavantage associé est qu'on doit stocker l'arbre et qu'on en fera autant de visite que d'analyse différente. Sur des ordinateurs moderne, cette approche n'est pas déraisonnable en terme de complexité.

En terme d'algorithmes impliqués, on aura donc pour chaque analyse, une fonction récursive réalisant un pattern-matching sur l'AST (donc un pour chaque type de l'AST)

qui prendra en entrée l'AST lui-même ainsi que des environnements, très similaire à ce que l'on avait pour l'interpréteur. La fonction décrira alors comment elle analyse le nœud courant de l'AST et comment elle se rappelle récursivement sur les (éventuels) enfants du nœud courant.

D'une manière générale, on peut voir ces analyses comme des formes d'interpréteurs qui au lieu d'exécuter le programme sur les vraies valeurs l'exécutent dans un autre ensemble et éventuellement le modifient. Ces analyses pourront modifier localement l'AST (lors des simplifications) et/ou faire des effets de bord sur les environnements passés en argument.

On manipulera une forme d'AST portant une annotation pour les expressions, instructions et déclaration de fonction (qui contiendra sa position dans le texte parsé, et des informations de type).

Dans notre cas, les annotations seront déjà placées dans l'AST, mais seule la position sera initialisée (par le parseur). On modifiera ces annotation par effet de bord lors de l'exploration de l'arbre dans les différentes analyses.

Enfin, on fera sur ces analyse du rapport des erreurs rencontrées, ce pour quoi chaque nœud de l'AST connaîtra sa position dans le texte analysé.

J'ai parlé de «famille de techniques» plus haut car si le pattern-matching est pratique pour cette idée, il n'existe pas dans tous les langages. En particulier, dans des langages orientés objets, tels que java ou C++, on pourra simuler cette approche grâce à un pattern de «visitor». N'ayant pas creusé la question, je n'en dirai pas plus, mais l'idée générale reste la même que ce qui est décrit plus haut.

4.2 Typage

L'une des analyses les plus importantes dans un langage typé statiquement (tous les langages ne le sont pas), c'est l'analyse de type. Elle consiste à déterminer pour chaque expression quelle est le type de cette expression, annoter l'arbre avec ce type, et détecter et reporter les erreurs au développeur.

Il existe principalement deux formes de cette analyse : la vérification de types et l'inférence de types, en fonction du langage sur lequel on travaille.

4.2.1 Vérification de types

Dans notre cas, on fait de la vérification de types. On est dans ce cas-là lorsque toute variable doit être déclarée avant d'être utilisées.

Dans un langage avec des types statiques basiques (ce qui est essentiellement notre cas), l'algorithme est relativement simple :

- Chaque déclaration de variable, de tableau modifie l'environnement en affectant un type à la variable concernée.
- Chaque déclaration de fonction crée un environnement de variable en typant ses paramètres, et ajoute son type à un environnement pour les fonctions.
- Chaque utilisation ou affectation d'une variable ou d'un tableau on interroge l'environnement pour déterminer son type, et on reporte une erreur si elle n'est pas déclarée.
- Les expressions composées (c'est-à-dire `Binop` et `Unop`) reçoivent le type lié à l'opération et reportent une erreur si leurs enfants n'ont pas des types cohérents (attention les comparaisons se comportent différemment des autres opérations).
- Chaque appel de fonction/procédure vérifie que le type des arguments est cohérent avec le type des paramètres, qu'aucun tableau n'est passé par valeur et que les

valeurs passées par référence sont bien des variables. On reporte une erreur si ce n'est pas le cas.

Néanmoins, les langages peuvent contenir des types plus complexes, par exemple les records en C (ou autres). Dans ce cas, chaque déclaration de type devrait être retenu et chaque utilisation d'un élément d'un record serait vérifié de la même manière que précédemment (il suffit de vérifier, par exemple que si on écrit `x.pouf`, alors le type déclaré pour `x` contient bien un champ `pouf`).

Dans le cas de notre langage, le seul cas similaire est le constructeur `Size` qui n'est défini correctement que si la variable pour lequel il est utilisé est un tableau.

4.2.2 Inférence de types

Il existe des langages pour lesquels la déclaration de types n'est pas obligatoire. C'est par exemple le cas de OCaml, mais aussi dans une certaine mesure du C++ (avec le type `auto`).

Dans ce cas, le compilateur doit inférer le type de chaque variable en fonction de comment elle est utilisée.

L'algorithme à utiliser est dans ce cadre plus complexe : il faut à chaque utilisation (ou éventuellement déclaration) d'une variable associer une contrainte de type, et ensuite résoudre un système de contrainte et reporter une erreur.

Dans le cadre d'un langage tel que le nôtre, cela serait relativement simple (les seuls types existants étant en nombre fini). Par exemple le code `x:= 1; x:= x +. 2.;` suffit à déterminer qu'il y a une erreur de type, puisque dans la première instruction, `x` est utilisé comme un entier et dans la seconde, comme un flottant. Similairement, `y := tab.size;` est suffisant pour déterminer que `y` est un entier et `tab` un tableau, bien qu'on ne sache pas à ce point ce que ce tableau contient.

Cependant, dans un langage contenant des records ou des fonctions typables, les types possibles sont plus complexes. Ils sont en général représentés comme des arbres, et chaque contrainte peut être vue comme un arbre avec des trous qui va contraindre le type considéré. En cas d'inconsistance une erreur est levée. Dans des langages avec des types générique, on peut même laisser des variables de types non complètes : en OCaml, c'est ce qu'on observe avec des types génériques. Dans un langage sans type générique, une erreur sera reportée si le type ne peut être déterminé uniquement.

On ne parlera pas ici de l'algorithme d'unification de type par manque de temps. Dans l'idée, il consiste à tenter d'unifier les arbres à chaque utilisation d'une variable.

Par exemple, dans un code du type `let x = y.pouf - y.bim`, l'inférence de type OCaml déterminera que `x` est un entier (à cause de `-`) et que `y` est un record contenant au moins les champs `pouf` et `bim` qui doivent être des entiers. C'est évidement compliqué par le fait que des champs de différents types peuvent avoir le même nom (mais l'inférence de type OCaml est l'une des plus puissante et ne fait pas l'objet de ce cours).

4.3 Analyse de portée

L'analyse de portée consiste à ce que les déclarations de variables n'aient un effet réel que dans le bloc où elles sont déclarées, de manière à avoir la possibilité de réutiliser des noms déjà existants.

Ainsi, le code suivant est tout à fait correct dans notre langage :

```
if (a = b) then{
    int x;
```

```
    x := 1;
    y := y + x;
}
else {
    float x;
    x := 2.3;
    z := z +. x;
}
```

En effet, les deux déclarations de `x` sont à l'intérieur de blocs différents et ne sont donc pas en conflit. Ce ne serait évidemment pas le cas si `x` était déjà déclaré dans un bloc contenant cette même instruction.

Pour cela, on va utiliser des environnement qui peuvent gérer des piles de déclaration : chaque entrée dans un bloc rajoute un niveau, chaque sortie d'un bloc retire un niveau, ce qui permet d'oublier les déclaration locale.

De la même manière, chaque déclaration de fonction crée un nouvel environnement de variable ne contenant que les arguments de la dite fonction.

Dans le cas où des variables globales existeraient (ce qui n'est pas notre cas, mais l'est en C), il faudrait évidemment les traiter différemment.

4.4 Initialisation

De la même manière, on peut également vérifier par le même genre de technique si une variable a déjà été initialisée lors d'une utilisation.

Dans ce cas, cela revient à chaque déclaration de fonction à retenir que la variable est déclarée mais pas initialisée, lors d'une affectation, à retenir que la variable affectée est initialisée, et lors d'une lecture, à lever une erreur si la variable est déclarée mais pas initialisée.

Il n'est pas simple de lier cette analyse d'initialisation à celle de portée, mais deux approches existent : une sûre (que l'on choisira) qui consiste à lever des avertissement si une variable n'a pas été initialisée dans un bloc contenant le bloc courant, et une moins sûre (comme en C) qui ne lève un avertissement que si la variable ne peut pas avoir été initialisée sur aucune des exécution menant à la position courante.

Dans notre programme, on liera les trois dernières analyse dans une seule fonction, mais vous pourrez les implémenter successivement.

4.5 Simplification de sucre et de constante

Toujours en appliquant les mêmes algorithmes, on peut simplifier les arbres obtenus vers en langage intermédiaire plus simple en :

— Simplifier les expressions constantes et instructions simplifiables.
— Éliminer des constructions «sucre» pour les remplacer par des constructions plus simples.

Dans notre langage, nous n'aurons que le premier cas. On y simplifiera deux types de constructions : les expressions constantes, par exemple `3 + 2` sera remplacé par `5`, et les `IfThenElse` où le test est constant (`if true then i_1 else i_2` sera remplacé par `i_1`) et les boucles dans lesquelles on n'entre jamais (`while false body` sera remplacé par `{}`).

Pour effectuer cette simplification, on parcours l'arbre de la même manière que dans les analyses précédente, mais au lieu d'annoter le terme avec le résultat, on renvoie le terme

simplifié (si on peut le simplifier) ou inchangé.

Le second cas correspond à des constructions qui sont autorisées dans un premier langage intermédiaire, mais que l'on remplace par une (ou plusieurs) constructions. Parmi les exemples classiques de ces remplacements, citons-en deux que l'on fait en général à ce niveau :

— En C, le for n'est pas un élément du langage intermédiaire. `for (A;B;C)body;` est remplacé à cette étape par `A; while(B){body; C}`.

— Dans plusieurs langages, on a des tableaux à plusieurs dimensions. On peut à cette étape les linéariser en des tableaux à une seule dimension (si on connaît leur taille). Cela dépend évidemment du langage (en C par exemple, ce n'est pas le cas, les tableaux à plusieurs dimension étant des pointeurs vers des pointeurs). L'idée est la suivante : considérons un tableau à deux dimension `int tab[s1][s2]` On le transforme en un tableau à une seule dimension de taille $s1 \times s2$ et on remplace `tab[i1][i2]` par `tab[s1*i1 + i2]`. Si le tableau a plus de deux dimensions, généralise cette transformation (on peut considérer qu'on l'applique plusieurs fois).

4.6 Erreurs et avertissements

Lors des passes précédentes (sauf la simplification qui ne peut pas réellement en contenir), on sera amené à détecter des erreurs dans le programme. Il convient de les reporter à l'utilisateur.

Pour cela, on pourrait simplement les afficher dans la sortie standard lorsqu'elles surviennent. Cependant, cela a le désavantage d'alourdir un peu le code et de ne pas être capable de déterminer à la fin de l'analyse si des erreurs ont été rencontrées.

On utilisera donc une structure permettant de retenir les erreurs et avertissement survenus. À chaque erreur ou avertissement, on ajoute un élément dans le rapport d'erreur contenant un message (une chaîne de caractères) et la position du terme où l'erreur est survenue (celle de l'annotation).

À la fin de l'analyse, on pourra alors aisément afficher toutes les erreurs et les avertissements séparément. La position permettra de les afficher dans leur contexte (c'est-à-dire le texte correspondant au terme reconnu). On pourra aisément déterminer la présence d'erreur pour, le cas échéant stopper la compilation à ce stade (s'il y a une erreur, exécuter ce programme n'aura pas de sémantique et est donc dangereux).

Enfin, si on ajoute un identifiant aux erreurs et aux avertissement, on peut également ignorer certaines erreurs/avertissements ou transformer des avertissements en erreur. Je vous renvoie vers le manuel de gcc ou du compilateur OCaml pour voir de quelles options ceux-ci disposent sur ce point.

Un point auquel faire attention est qu'il est préférable d'éviter qu'une même erreur soit reportée plusieurs fois. Typiquement, si une variable n'est pas déclarée, en général elle est utilisée plusieurs fois, et donc cela mènerait à des messages redondants. Pour cela, une technique consiste (dans ce cas) à attribuer un type spécial à une variable dont on détecte qu'elle est non-déclarée lors de sa première utilisation et d'ignorer ce type dans les rapport d'erreur. Dans notre cas, on pourra utiliser `TNull` pour cela.

Chapitre 5

Code à trois adresses et génération de code naïve

Jusqu'ici, on a tout ce qu'il faut pour faire un interpréteur complet, qui rejette les programmes incorrects. Pour un interpréteur, cela suffit (on peut continuer à optimiser les parties précédentes, mais on a tout les ingrédients). Pour un compilateur, il manque maintenant la traduction du code vers l'assembleur, gestion de mémoire incluse, ainsi que l'optimisation du code ainsi généré.

Il est possible – et assez traditionnel de casser cette étape en plusieurs partie :

— Une traduction vers de l'assembleur de haut niveau, où les instructions correspondent à celles de l'assembleur, mais où la gestion de la mémoire n'est pas effectuée. Ce code s'appelle généralement «code à trois adresses».

— Une phase d'optimisation du code produit, en coupant le code en bloc, en analysant le flot du programme, en optimisant chaque bloc indépendamment et en les réordonnant.

— La traduction vers l'assembleur final, en sélectionnant les instructions exactes, en gérant explicitement la mémoire (affectation des registres) et en appelant les routines exactes du système vers lequel on compile.

Seule la dernière passe est réellement dépendante de la machine vers laquelle on compile.

Tous les compilateurs n'utilisent pas nécessairement une telle représentation intermédiaire – cela peut varier : il est possible de faire directement la traduction vers l'assembleur et optimiser ensuite. Il existe également des frameworks génériques, comme LLVM, qui visent à fournir les outils pour ces passes.

Néanmoins, cette découpe permet de séparer différentes transformations assez différentes et présente donc au moins un intérêt pédagogique.

Dans ce chapitre, on se concentre sur le code à trois adresse et la traduction de l'AST annoté vers celui-ci.

5.1 Code à trois adresses

Le code à trois adresses n'est pas un langage défini universellement, il y a autant de tels langages que de compilateurs qui en utilisent et de cours qui le présentent. Dans ce cours, on prend un langage très fortement inspiré de celui présenté dans [1], à quelques différences près pour l'adapter à nos besoins :

— On conserve comme instructions des routines d'affichage et de déclaration mémoire. Usuellement, on utiliserait des appels de fonctions, mais ici cela facilite l'interpré-

tation. En effet, comme on s'arrête dans ce cours à cette phase là, on fournit un interpréteur, pour que vous puissiez tester votre traduction.

Le code à trois adresses prend son nom du fait que les seules opérations sont des opérations unaires et binaires qui prennent deux adresses (ou constantes) et écrivent dans une troisième adresses. Chaque instruction manipule donc au plus trois adresses mémoires. Ces opérations correspondent aux opérations du processeur.

Les seules instructions permettant le flot de contrôle sont des sauts conditionnels et inconditionnels, ainsi que des instructions d'appel de fonction et de retour de fonction.

Si la mémoire n'est pas explicitement gérée, elle est tout de même ici séparée en deux : les adresses, qui ici sont des noms simple mais en assembleur seront stockées sur la pile du programme et dans les registres, et les tableaux, qui dans l'assembleur seront stockées dans le tas et ne peuvent directement apparaître comme opérande d'une opération.

On peut maintenant donner le code à trois adresses que nous manipulons.

Une adresse est une chaîne de caractères (bien qu'elle soit abstraite dans notre implémentation). On note son type `address`.

Une *right value* est soit une adresse, soit un entier, soit un flottant, soit un booléen. Son type est :

```
type r_value =
  | Name of address
  | Int of int
  | Float of float
  | Bool of bool
```

Ces valeurs représentent ce qui peut apparaître comme opérande d'une opération ou d'une affectation. Elles restent typées par les types de bases pour des raisons techniques : l'écriture du code assembleur exact gérera la bonne écriture (qui est différente en fonction du type en question), et cela, comme plusieurs autres choix, facilite l'interprétation de ce code.

Une adresse de tableau est un couple contenant une adresse et une right value : `type array_address = address * r_value`.

Les opérateurs unaires et binaires correspondent à celles de l'AST (on ne les recopie pas ici, vous vous reporterez au code), à la différence près qu'on a des opérateurs spécialisés pour les entiers et les flottants.

Les *étiquettes* (ou label) sont des chaînes de caractères qui étiquettes certaines instructions du programmes. Elles seront les destination des saut.

Enfin les instructions du code à trois adresses sont les suivantes :

```
type instruction =
| Binop of address * binop * r_value * r_value
| Unop of address * unop * r_value
| Copy of address * r_value
| ADecl of address * r_value
| ACopy of array_address * r_value
| ALoad of address * array_address
| AddressOf of address * address
| SetPointed of address * r_value
| GetPointed of address * address
| Goto of label
| IfTrue of r_value * label
| IfFalse of r_value * label
```

```
| Param of address
| Call of label * int
| Noop
| Return
| Print_int of r_value
| Print_float of r_value
| Print_bool of r_value
| Print of string
```

Commentons chaque construction :

— `Binop` et `Unop` réalisent l'opération correspondante sur les `r_value` en argument et placent le résultat dans l'adresse fournie.
— `Copy` copie la `r_value` dans l'adresse fournie. Cette instruction est nécessaire dans certaines traductions que l'on verra plus tard, mais est une bonne candidate à l'optimisation.
— `ADecl` est une instruction qui correspondra à un appel de fonction système. C'est un malloc qui ira réserver un tableau de la bonne taille sur le tas. Elle est ici laissée abstraite.
— `ALoad` et `ACopy` sont des instructions de lecture/écriture dans le tableau. Étant donné que ce sont des zones mémoires non-directement accessibles au processeur, on doit copier leurs valeurs dans des adresses avant de les utiliser.
— `AddressOf`, `SetPointed` et `GetPointed` sont des instructions permettant de manipuler des pointeur, le premier en récupérant l'adresse où est stockée une adresse, le second en affectant une valeur à la zone pointée à une adresse, la dernière en récupérant la valeur stockée à une adresse. Ces trois constructeurs seront réservés dans notre cas à la gestion des variables passées par paramètre (bien que dans un cas général, on les utiliserait pour les pointeurs, évidemment).
— `Goto` est un saut inconditionnel vers le label en argument.
— `IfTrue` et `IfFalse` sont des saut conditionnels (en fonction de la valeur de l'expression) vers le label en argument.
— `Noop` ne fait rien (et est utile pour simplifier la génération de code, notamment les labels).
— Les instructions `Print` sont des routines d'affichages typées – qui correspondraient à des appels système et sont ici abstraits pour faciliter l'interprétation.
— `Param`, `Call` et `Return` sont les instructions qui gèrent les appels de fonctions. `Param` empile une adresse comme argument de la prochaine fonction à être appelé. `Call` empile une adresse de retour, dépile les paramètres correspondant à l'appel (les n derniers, si n est l'argument) et saute au label correspondant à la fonction. `Return` revient à l'adresse de retour précédente (empilée par `Call`). Les paramètres étant gérés comme une pile, il est possible d'avoir des appels de fonction imbriqués.

Comme l'interpréteur de ce code vous est fourni, je ne donne pas ici de sémantique formelle dans cette version du document (extension possible plus tard). Dans les grandes lignes, la machine abstraite a une seule mémoire, séparée en trois parties étanche : celle contenant les adresses, celle contenant les tableaux et celle contenant le code, ainsi qu'un pointeur de programme, une pile d'adresses de retour et une pile d'arguments. Chaque instruction effectue ce qui est décrit plus haut, puis fait avancer le pointeur de programme de 1 (sauf les sauts).

5.2 Traduction vers le code à trois adresses

On peut maintenant décrire dans les grandes lignes la traduction de notre AST vers le code à trois adresses, en le faisant étape par étape.

5.2.1 L_{calc}

Dans le sous-langage L_{calc}, il n'y a pas de difficulté particulière.

Pour les expressions, il s'agit de simplement découper l'expression en opérations de base et à stocker chaque sous-résultat dans une adresse fraîche (c'est-à-dire créée pour l'occasion). Cela se définit récursivement comme suit :

— Les constantes sont traduites par elles-même.
— Les variables sont traduites par une adresse de même nom (sauf si ce sont des paramètre de la fonction courante, cf plus bas).
— Pour une opération binaire `Binop(op,e1,e2)`, on commence par traduire `e1` et `e2` en code trois adresses, et en supposant que `n1` et `n2` sont les right values correspondant aux résultats de ces deux expressions, on ajoute `Binop(a3,op,n1,n2)` au code trois adresses et la right value correspondant à cette expression est `Name a3`.
— Les opérations unaires sont traitées similairement.

Les affectations correspondent directement à des `Copy`, et pour un `Block`, on traduit simplement successivement toutes les instructions qu'il contient.

La déclaration d'une variable n'a aucun effet.

Il y a tout de même une subtilité quand on va avoir des appels de fonctions (voir plus bas) : la gestion des arguments passés par référence. Comme il apparaîtra plus bas, la solution choisie pour ceux-ci est de passer leur adresse, et non leur valeur. Il faut donc, pour les utiliser, les utiliser comme des pointeurs. Aussi pour ceux-ci, il faudra utiliser les instructions `GetPointed` (pour les variables (i.e., lectures)) et `SetPointed` (pour les affectations). Le module `Arg_mapper` vous aidera à retenir les informations nécessaires, en connaissant un nom désignant l'argument, et en ayant retenu s'il est passé par valeur (auquel cas on le traite comme une variable locale), ou par référence.

5.2.2 L_{array}

Les tableaux ne présentent pas non plus beaucoup de difficulté :

Les instructions de lecture, écriture et déclaration d'un tableau correspondent directement à `ALoad`, `ACopy` et `ADecl`.

Pour `Size_tab`, on considère que pour chaque déclaration de tableau `tab` on affecte la taille à la case -1 du tableau.

En pratique, lors de la gestion de la mémoire dans le passage à l'assembleur, on forcera ce comportement en attribuant à l'adresse contenant l'adresse du tableau une case plus loin que la réservation effective. Ici, `ADecl` se charge de ce détail technique.

5.2.3 L_{branch}

Pour les structures de contrôle, la traduction devient un peu plus subtile.

Pour le `IfThenElse(test,i_then,i_else)`, on traduit indépendamment `test`, `i_then` et `i_else`, et si on nomme `n_test` la right value où la valeur du test est stockée, on produit simplement le code suivant :

```
[code_test]
IfFalse(n_test,Lbl_Else)
```

```
[code_i_then]
Goto Lbl_End
Lbl_Else: Noop
[code_i_else]
Lbl_End: Noop
```

Pour le `While(test,body)`, le principe est le même, on traduit indépendamment `test` et `body` et on produit le code suivant :

```
Lbl_test: Noop
[code_test]
IfFalse(n_test,Lbl_End)
[code_body]
Goto(Lbl_test)
Lbl_End : Noop
```

À noter que dans le cas d'un do while, la traduction échangerait simplement test et body dans la traduction (et aurait un `IfTrue`).

5.2.4 $L_{function}$

C'est pour les fonctions, comme toujours, que la traduction est la moins immédiate. Le principe général en est cependant simple.

Si on doit traduire `Call(name,[e1;...;ek])`, on doit d'abord traiter les arguments et les empiler avec l'instruction `Param`, réaliser l'appel, puis récupérer le résultat.

Pour traiter les arguments, il faut le faire en commençant par la droite et non par la gauche : cela permet d'avoir le premier argument comme sommet de pile et donc de traiter plus aisément les arguments dans la fonction appelée. C'est pour cette raison que dans de nombreux langages (et le nôtre également), les arguments sont évalués en commençant par celui de droite. Ensuite, le traitement dépend de si l'argument est passé par valeur ou par référence. Pour l'argument `ei` s'il est passé par référence et que l'adresse où il est stocké est `add` (ce ne peut pas être une constante par construction du langage), on crée une adresse fraîche `ai` et on traduit cela en :

```
[code_ei]
AddressOf(ai,add)
Param ai
```

Si au contraire `ei` est passé par argument et que la right value lui correspondant est `ni` et que ce `ni` n'est pas une adresse, on créée une adresse fraîche `ai` et on le traduit en

```
[code_ei]
Copy(ai,ni)
Param ai
```

Si `ni` est une adresse, on peut la passer directement :

```
[code_ei]
Param ni
```

Il n'est pas nécessaire de rendre cette adresse indépendante, puisque c'est précisément ce que fera l'instruction `Param` (créer une pile d'appel).

Pour une vraie traduction vers l'assembleur, il serait plus efficace de séparer les instructions `Param` du reste pour avoir une simple séquence d'instructions `Param` d'un coup,

dont on est certain qu'elle n'interagira pas avec les traduction de `ei`. C'est un détail qui ne sera pas important ici.

Une fois ce traitement des arguments effectué, on réalise l'appel simplement avec `Call(name,k)`.

Il faut sur l'instruction suivante (qui sera l'adresse de retour), stocker le résultat dans une valeur fraîche. Le résultat de la fonction sera, par convention stockée dans une adresse de nom `#result#` qui ne fait pas partie des noms autorisés pour des variables dans le langage.

Les appels de procédures sont exactement similaires, à part que la valeur de retour n'est pas utilisée.

L'instruction `Return` est traduite en `Return` directement. Si elle contient une expression, cette expression est au préalable traduite et la right value obtenue est stockée dans l'adresse `#result#` avant l'instruction `Return`.

Enfin, une déclaration de fonction `Func_decl (typ, name, params, body, annotation)` est traduite en plaçant au début le label `name`, et en traduisant ensuite `body`, dans lequel toutes les adresses faisant partie de `params` sont remplacées par une adresse de nom `#arg_i#` où i est leur position dans la liste `params`. Vous aurez une aide pour le code de ce point, mais l'idée est d'être capable d'identifier clairement leur position dans la pile de paramètres. On fait suivre la traduction de `body` par une instruction `Return` pour éviter les cas où une fonction ne terminerait pas par un `Return` dans le code à compiler. C'est dans le traitement de cette déclaration qu'on créera une valeur de type `Arg_mapper.t` et qu'on la remplira avec la liste des paramètres.

Enfin, pour traduire un programme complet, on traduit simplement successivement toutes les déclarations de fonctions du programme, et on fait précéder le tout d'une instruction `Goto main` pour sauter au point d'entrée du programme.

5.3 Pour aller plus loin

Pas eu le temps, mais à compléter plus tard.

Bibliographie

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[2] Clinton L. Jeffery. Generating LR syntax error messages from examples. *ACM Trans. Program. Lang. Syst.*, 25(5) :631–640, 2003.

[3] Donald E. Knuth. On the translation of languages from left to right. *Inf. Control.*, 8(6) :607–639, 1965.

[4] David Pager. A practical general method for constructing lr(k) parsers. *Acta Informatica*, 7 :249–268, 1977.

[5] François Pottier. Reachability and error diagnosis in LR(1) parsers. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 88–98. ACM, 2016.