Projet de recherche lambda.x

Laboratoire I3S
Informatique, Signaux et Système de Nice - Sophia-Antipolis

Etude préalable de D.E.A.

- 1992-1993

Sujet :

- λ-calcul,
- langages fonctionnels,
- graphes Data Flow,
- interpréteur Scheme.

Sous la responsabilité de :

- Professeur Fernand Boéri,
- Professeur Jean Demartini

Etudiant :

- Guilhem de Wailly

Laboratoire d'informatique Signaux Système

❑ 41 Bd Napoléon III - 06041 Nice cedex France. Tel.(33) 93 21 79 56
❑ Bat. 4 - 250 Av Albert Einstein - Sophia Antipolis - F 06560 Valbonne

Rédigé en mai-juin 93 sur traitement de texte Word© 2.0c pour Windows© 3.01 de MicroSoft®

PREFACE

Ce rapport et le résultat du stage effectué dans le laboratoire I3S de Nice-Sophia Antipolis dans le cadre du D.E.A. T.T.I. Ce stage sanctionne le diplôme de fin d'année. Il se doit d'être un stage de recherche, mais aussi un stage de réalisation.

Je tiens à remercier MM. Boéri et Demartini, mes directeurs de stage, qui m'ont apporté leur soutient constant. Leur écoute permanente, leur critiques constructives m'ont permis de progresser en me remettant souvent en cause. Ils m'ont permis de prendre goût à la recherche.

M.Augin m'a accueillis dans le laboratoire dont il a la charge et m'a fait entière confiance pour mener à bien ce projet. Je lui en suis reconnaissant et redevable.

Enfin, tous les membres du laboratoire m'ont accueilli chaleureusement et m'ont aidé aussi souvent que j'en avais besoin. Qu'ils en soient remerciés.

Merci à tous les auteurs que j'ai cités tout au long de ce rapport, sur lesquels je me suis continuellement appuyé pour progresser.

TABLE DES MATIERES

Préface 3
Table des Matières 4
Symboles & Conventions 6
Tête de Chapitre 6
Titre de niveau 3 6
Titre de niveau 4 6
Titre de niveau 5 6
Introduction 7
Projet de Recherche 8
Renseignements administratifs 9
Titre 9
Objet 9
Analyse des travaux et Motivations 10
Situation générale et originalité 10
Présentation sommaire des travaux proposés 10
Phase 1 : étude de la modélisation fonctionnelle d'une application sous la forme d'une λ-expression 11
Structure d'arbre abstrait 12
Phase 2 : interpréteur de λ-expressions 12
Phase 3 : construction d'un graphe DFS 12
Aboutissement et prolongement des travaux 14
Construction de l'arbre abstrait d'une application parallèle et du graphe DFS associé 14
Conception d'une architecture à partir de son graphe DFS 14
Objectifs 15
Produit final 16
Aspect matériel 16
DSP 16
Contrôle du système 16
Entrées/sorties 17
Utilisation en monoprocesseur 17
Configuration multiprocesseur 18
Modes d'adressage et instructions 19
Résumé 19
Parallélisme 19
Parallélisme asynchrone 20
Parallélisme synchrone 20
Aspect logiciel 20
Graphes Data Flow 21
Langages fonctionnels 21
PARTIE 1: Langages Fonctionnels 23
CHAPITRE 1: Le λ-Calcul 24
Qu'est que le λ-calcul 24
Syntaxe du λ-calcul 25
Application et curryfication 25
Parenthèses 26
λ-abstraction 26
La sémantique du λ-calcul 27
Variables libres et liées 27
β-conversion 27
Exemples de β-réductions 28
Noms de variables 28
Un exemple 29
Conversion, réduction et abstraction 29
α-conversion 29
η-conversion 30
Capture des noms de variables 30
δ-conversion 31
Résumé 31
Ordre de réduction 31
Réduction en ordre normal 32
Ordre de réduction optimal 33
Fonctions récursives 33
Fonction récursive Y 34
Y est une λ-expression 35
Sémantique dénotationnelle du λ-calcul 35
La fonction Eval 36
Notation 37
Le symbole ⊥ 37
Constantes & fonctions pré-définies 38
Caractère strict et paresseux 39

Règles de conversion....39
Egalité et convertibilité....40
C H A P I T R E 2: Langages Fonctionnels....41
Lisp....41
Iswim....42
Lucid....43
Scheme....43
Eléments de base....44
Type des données....44
Appel de function....44
Arguments....44
Définitions....45
Formes spéciales....45
Durée de vie des objets....45
Evaluation des expressions....45
Listes....46
Représentation externe....47
Fonctions....47
Fonctions : des objets comme les autres....48
CONS, CAR et CDR en Scheme....49
Variables libres, liées et environnements.....49
Fonctions récursives....50
Récursion de queue....52
Action sur les environnements....52
Define....52
Set!....53
Let....53
Letrec....54
Let*....55
Autres fonctions....55
Evaluation de Scheme....56
PARTIE 2: Les graphes Data Flow....57
Modèles Théoriques....58
Théorie des ordinateurs....58
Von Neumann....58
Modèles Data Flow....59
Graphe simple....59
Contre-réaction....60
Valeur initiale....61
Data Flow synchrone....61
IOR....62
Horloge....62
Opérateur de recyclage....63
Conclusion....64
PARTIE 3: Interpréteur & compilateur SCHEME....65
Outils De Base....66
Atomes....66
Gestion de la mémoire....67
Le tas....68
La pile....68
Le garbage....68
Vecteurs....69
Table des symboles....69
Interface avec le langage c....70
Appel de fonction....70
Pile....70
Type de fonction....71
Interprétation & Compilation....72
Analyse syntaxique....72
Evaluateur....73
Compilation....73
Lambda....74
Structure de donnée des λ-expressions....74
Compilation....75
Let....76
Extension De Scheme....77
Fichiers compilés....77
Librairies à liens dynamiques....77
Synthèse....79
Bibliographie....82
Index....85

SYMBOLES & CONVENTIONS

Nous présentons ici les symboles et convention utilisées dans ce rapport.

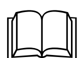
Indique une proposition juste ou valide.

Indique une proposition fausse ou erronée.

Indique une proposition importante. En général elle est restrictive.

Indique une proposition importante.

Indique une proposition dont il faut se rappeler.


Indique une référence bibliographie qui figure ou non dans la bibliographie.

Indique un commentaire, une précision.

Le texte du rapport est dans cette police de caractère. Les anglicismes apparaissent en *italique*. Les exemples, le code dans un langage informatique

```
apparaissent comme cela.
```

TETE DE CHAPITRE

Titre de niveau 3

Titre de niveau 4

Titre de niveau 5

INTRODUCTION

Le présent rapport s'inscrit dans le cadre d'un D.E.A. Il est une étude préliminaire et devrait servir de base à une thèse. Son but est de découvrir un certain nombre de concepts et de méthodes et d'ouvrir des pistes. Il représente le travail d'une année, répartie en deux phases, l'une de recherche bibliographique, et l'autre d'implémentation. Ces phases ne sont pas réparties équitablement dans le temps, puisque la première dura deux tiers de l'année.

Ce rapport aborde le domaine des langages fonctionnels, la théorie des graphes Data Flow, et l'implémentation d'un langage fonctionnel. Nous ne saurions prétendre avoir une connaissance approfondie de chacun des sujets, mais plutôt une vision globale et générale.

Le projet de recherche est présenté dans le prochain chapitre tel qu'il fut communiqué à la D.R.E.T. Ainsi, nous avons constamment à l'esprit le but à atteindre.

PROJET DE RECHERCHE

Le projet de recherche fait l'objet d'un contrat DRET. Il a été rédigé par MM F.Boéri et J.Demartini. Nous avons choisi de l'inclure dans ce document pour justifier des choix que nous avons fait, gardant en mémoire l'objectif final.

Renseignements administratifs

Titre

Prononcer lambda X

La nature du travail entrepris ainsi que les outils théoriques utilisés nous ont amenés à appeler ce projet :

λX

Objet

Une application est complexe lorsqu'elle doit mettre en jeu de nombreux éléments ayant de nombreuses interactions entre eux.

Etude et définition d'une approche destinée à l'analyse et à la structuration d'une application complexe en général dans le domaine du traitement de signal. Cette méthode doit permettre d'une part de mettre naturellement en évidence le parallélisme sous-jacent de l'application et d'autre part aboutir à la conception matérielle adaptée.

Cette méthode permettrait de construire une représentation abstraite de l'application cible de telle sorte qu'on puisse la coupler ultérieurement à un outil ou à une méthode de conception d'architecture matérielle. Cette construction est systématique et assure l'équivalence entre l'expression des spécifications du problème et le résultat de la conception.

Dans une première phase, cette méthode est conçue pour aboutir à une architecture multi-DSP en utilisant une description Data Flow des opérations.

DSP = Digital Signal Processor

(...)

ANALYSE DES TRAVAUX ET MOTIVATIONS

Depuis plusieurs années, par le biais de plusieurs contrats avec la DRET, notre équipe à développé :

- des modèles de calculateurs parallèles de type SIMD et SPMD (OPSILA)
- des méthodes de simulation d'architecture (projet SAM-1 et SAM-2 [7,18] avec SIMULOG)
- une méthode de génération de simulateurs (LUXIFERE [15])
- une méthode de conception assistée de processeurs synchrones spécialisés (CAPSYS-1 et CAPSYS-2 [8,10])

Dans tous ces cas, nous utilisons une architecture de type Von Neuman pour le matériel et des langages de blocs pour modéliser l'application.
La présente étude s'oriente vers l'introduction de langages fonctionnels, de schémas fonctionnels ou d'équations pour décrire l'application à réaliser et des architectures Data Flow Synchrones, Data Flow distribués ou VLIW pour la réalisation matérielle.

Situation générale et originalité

Notre étude se situe dans le domaine de la synthèse dite de haut niveau (*System level Synthesis*), plus précisément dans le domaine de la spécification de système de traitement numérique du signal par une approche fonctionnelle et une synthèse par système multi-DSP ou VLIW (travaux futurs).
Plus généralement, nous proposons une méthode qui permettrait, pour la réalisation d'une application en traitement numérique du signal, l'utilisation de circuits intégrés complexes existants (DSP) ou conçus à partir d'outils de synthèse tels que TRANSE[10], GABRIEL[2] et CAPSYS[8,10].

Présentation sommaire des travaux proposés

La description et la spécification du comportement d'une application complexe (logiciel et/ou matérielle) fait intervenir actuellement de nombreux outils de représentation et de raisonnement. Ces outils correspondent à des niveaux d'abstraction différents associés à des stades différents de la spécification.
L'expérience a montré qu'il était souhaitable de rendre cette description / spécification aussi indépendante que possible de l'implémentation ultérieure qui en sera faite. Elle a montré également que les outils les plus sûrs étaient ceux pour lesquels il existait un fondement théorique solide.
Parmi tous les outils disponibles, on peut citer :

- équations et fonctions mathématiques,
- grammaires,
- machines à états finis,

- machines à pile,
- objets,
- types abstraits de donné,
- algorithmes.

Les quatre premiers éléments de cette liste sont des outils dont la théorie est bien maîtrisée. Par contre les trois suivants correspondent à des situations d'une trop grande généralité pour pouvoir être appréhendée en utilisant une théorie commodément utilisable. Dans le cadre d'une application complexe, on rencontre, en général, l'utilisation de tous ces outils en même temps car aucun d'eux ne permet d'assurer une description adéquate de tous les aspects de l'application.

Le projet λX ne prétend pas remplacer tous ces outils par une méthodologie universelle permettant une approche générale des problèmes de conception, mais simplement définir un moyen de confiner tous ces outils au niveau d'abstraction qui leur convient le mieux.

Le projet λX se déroulerait en trois phases :

- modélisation fonctionnelle directe d'une application de type traitement du signal,
- élaboration d'une technique de transformation de ce modèle vers une représentation sous la forme d'un arbre abstrait,
- transformation de l'arbre abstrait précédent en graphe Data Flow Synchrone (DFS).

Un graphe Data Flow est dit synchrone lorsque ses opérateurs consomment toujours le même nombre de données et produisent toujours le même nombre de résultats

Le graphe DFS est le point de départ d'une méthode de réalisation d'applications parallèles de traitement de signal utilisant un réseau de DSP. Cette méthode a été développée à l'Université de Berkeley par Lee et Messerschmitt [2]

.

Phase 1 : étude de la modélisation fonctionnelle d'une application sous la forme d'une λ-expression

Lorsqu'un ingénieur aborde la description d'une application complexe, sa première ébauche est en général un dessin. Ce dessin tente de dégager les grandes lignes du problème.

Le génie logiciel utilise largement cette remarque préliminaire lorsqu'il introduit des méthodes de conceptions du type SADT, SART ...

Ce schéma est souvent qualifié de fonctionnel, il est généralement hiérarchique et met en présence des boîtes et des traits. Les boîtes matérialisent des traitements d'information et les traits des flots d'information.

Dans le cas des applications de traitement du signal, les informations manipulées sont peu structurées et la conception est plutôt guidée par les traitements. Chaque traitement peut être vu comme un filtre opérant sur des flots d'entrées pour produire un flot de sortie.

Dans un tel contexte, l'aspect fonctionnel devient prépondérant. La représentation et la manipulation d'entités fonctionnelles sont actuellement essentiellement faites en utilisant le λ-calcul.

Une première phase du travail consiste donc à définir sous quelles conditions les outils de description et de spécification indiqués peuvent être formalisés à partir du λ-calcul et comment trouver la λ-expression équivalente à une description donnée.

Les outils de description / spécification fonctionnelle peuvent être :

- un système d'équations mathématiques,
- un programme fonctionnel écrit en utilisant un langage fonctionnel (applicatif) - SASL, LUCID, MIRANDA, LUSTRE ...,
- un schéma fonctionnel de type flot de données.

Toutes ces descriptions peuvent être unifiées sur la base du λ-calcul et transformées en λ-expressions. En ce qui nous concerne, nous avons une nette préférence pour une représentation par schéma fonctionnel hiérarchisé réservant les équations mathématiques ou les langages applicatifs à quelques descriptions ponctuelles.
L'objectif de cette première phase du projet est de déterminer dans quelles conditions on peut mélanger toutes ces formes de description afin de pouvoir utiliser librement celle qui convient au moment qui convient.

Structure d'arbre abstrait

La λ-expression trouvée précédemment représente toutes les dépendances fonctionnelles qui existent entre les différents éléments de la description, la hiérarchie sous-jacente ainsi que le parallélisme qu'il est possible d'extraire.

Une compilation de cette λ-expression, qui la mettra en forme normale, effectue une mise à plat de la description.

Cet arbre abstrait est un arbre binaire dans lequel chaque noeud représente l'application fonctionnelle de son fils gauche à son fils droit.

Cette forme normale peut être mise sous la forme d'un arbre abstrait. Cet arbre abstrait est utilisé d'habitude pour construire un exécuteur du programme correspondant sous la forme d'un code séquentiel, d'une machine à réduction ou d'une machine Data Flow.
Cet arbre abstrait peut servir de base pour la conception d'une architecture logicielle ou d'une architecture matérielle d'exécution spécialisée de ce programme.
La compilation ainsi effectuée consiste en une séquence de transformation du programme initial (spécification) vers sa forme compilée (forme normale). Ces transformations garantissent l'équivalence de la forme initiale et de la forme finale.

Phase 2 : interpréteur de λ-expressions

L'approche décrite suppose évidemment que les spécifications sont correctes. Par contre, cette approche garantit au concepteur qu'il n'introduit pas d'erreurs de conception.

Jouer les spécifications d'une application revient ici à interpréter une λ-expression.

Le développement d'un compilateur-interpréteur de λ-expressions constitue la deuxième phase du projet.

Phase 3 : construction d'un graphe DFS

Le mot processeur doit être pris au sens large. Il ne s'agit pas nécessairement d'un composant, ce peut être une carte complète, un système complet.

Une fois que l'arbre abstrait de l'application à été obtenu, il peut être utilisé pour concevoir l'architecture matérielle et logiciel d'un processeur spécialisé. A ce niveau, plusieurs choix sont possibles.
Dans cette proposition, l'architecture matérielle choisie est un réseau de DSP standards fonctionnant en parallèle. Le problème posé peur être schématiquement résumé ainsi : comment programmer les DSP disponibles sur la carte utilisée pour implémenter l'application cible ?
L'arbre abstrait défini ci-dessus doit être conçu pour permettre sa transformation en graphe DFS qui constitue le point d'entrée de la méthode de conception d'application parallèle élaborée par Lee et Messerschmitt de l'Université de Berkeley[2]. Cette adaptation constitue la troisième phase du projet.

ABOUTISSEMENT ET PROLONGEMENT DES TRAVAUX

Construction de l'arbre abstrait d'une application parallèle et du graphe DFS associé

L'aboutissement de ces travaux est la réalisation de l'environnement de développement de l'arbre abstrait d'une application décrite sous forme fonctionnelle.
Cet environnement comporte trois partie :

- outils de description d'applications fonctionnelles sous la forme de schémas fonctionnels, d'équations mathématiques ou de programmes écrits en utilisant un langage applicatif à définir. Cet outil se présentera naturellement sous la forme d'une application graphique interactive,
- compilateur / interpréteur de λ-expressions,
- générateur de graphe DFS.

Conception d'une architecture à partir de son graphe DFS

L'arbre abstrait de l'application peut être transformé en un graphe Data Flow Synchrone. Les opérateurs de ce graphe peuvent être ensuite répartis sur un réseau de DSP en utilisant l'approche développée par Lee et Messerschmitt de l'Université de Berkeley [2].
Sommairement, l'approche DFS pour la conception peut être résumée ainsi :

- le graphe DFS décrit les dépendances fonctionnelles entre les différents opérateurs utilisés pour la réalisation de l'application cible,
- ce graphe est scindé en autant de sous graphes qu'il y a de DSP disponibles,
- chaque sous graphe est transformé en programme séquentiel en déterminant le séquencement admissible de ce sous graphe,

L'implémentation de cette approche dans le cas d'une application test pourrait constituer la suite de ce projet.

OBJECTIFS

Nous présentons ici nos plans pour parvenir à remplir le cahier des charges. Ce cahier des charges est imprécis quant aux moyens à utiliser tout en ouvrant quelques voies. C'est que le sujet est vaste. En effet, depuis plusieurs années, de nombreuses équipes de recherche travaillent dans ce sens. Il existe un certain nombre de machines parallèles, mais elles sont souvent très spécialisées, comme TerraData, spécialisée dans la gestion des bases de données, HyperCube, dans le traitement des images. En ce qui concerne le traitement du signal, peu d'applications ont vu le jour. Le champ d'investigation est donc assez vaste, le projet de recherche ne doit pas être restrictif.

Dans le chapitre qui suit, nous présentons le produit final tel qu'"il faudrait qu'il soit". Cette présentation est bien sommaire, mais elle permet de fixer certains jalons sur la route à suivre.

ABOUTISSEMENT

Ce projet se propose de fournir au terme de la thèse qui suivra ce rapport un logiciel "clef en main", opérationnel. L'objectif est donc une implémentation "physique" qui va se traduire par un certain nombre de ligne de code dans un certain langage et pour certains ordinateurs.
Il peut paraître étrange de présenter ici l'aboutissement des travaux alors que le travail est à peine commencé. Mais il nous semble que d'imaginer ce qu'il sera à la fin, et cela du point de vu de l'utilisateur, donc du point de vue fonctionnel, permet d'avoir une approche plus générale des moyens à mettre en oeuvre.
La théorie doit être au service du produit final, et non le contraire. L'étude des travaux des autres équipes qui travail sur des projets similaires doivent nous indiquer les voies à suivre et les écueils à éviter.
On peut considérer cet aboutissement en deux parties, l'aspect matériel, et l'aspect logiciel.

Aspect matériel

D'un point de vu matériel, le but poursuivi est de fournir une architecture d'ordinateur orienté dans le traitement des données numériques fortement parallèle. Cette architecture se traduira soit par une carte électronique soit par un modèle destiné à un émulateur.
Le traitement numérique de donnée nous incite à utiliser les D.S.P. ou *Digital Signal Processor*, processeurs spécialisés en la matière.

DSP

Nous présentons ici à titre d'exemple le DSP de Texas Instrument TMS 320C25. Il ressemble par ses possibilités à la plupart des DSP actuels du marché.

Contrôle du système

Ce DSP possède un *timer*, un compteur de répétition, trois interruptions masquables et externes et une interruption générée par le *timer* ou le port série.
Le *timer* décompte continuellement l'horloge ; lorsque celle-ci atteint zéro, l'interruption *timer* est générée et le *timer* est rechargé avec le registre *period*. De cette manière on peut programmer la période de l'interruption à sa guise.
L'instruction de répétition d'opération autorise 256 exécutions successives d'une même instruction. Le compteur peut être initialisé soit à partir de la mémoire des données soit par une valeur immédiate provenant de la mémoire du code.
Trois interruptions internes déroutent le programme en cours vers des routines d'interruption. Elles sont générées par le *timer*, le port série ou le programme en cours d'exécution. Il leur est attribué une priorité.
Deux registres sont utilisés pour sauver le contexte d'exécution lors d'un appel à un sous-programme ou à une interruption.

Entrées/sorties

Dans une utilisation de plusieurs DSP en parallèle, cela permet de gérer facilement des zones mémoires d'échanges communes.

Le TMS 320C25 est capable d'interfacer une très grande étendue d'adresses du fait de ses trois champs d'adresses distincts : le champ programme, le champ données et le champ entrées/sorties.
Les entrées/sorties sont traitées de la même manière que les données. L'utilisation des bus d'adresses et données externes au DSP est gérable de la même manière pour les données mémoires et les entrées/sorties.

Utilisation en monoprocesseur

Ce DSP permet aisément d'implémenter un système minimum. Il possède un bus de données et un bus d'adresses de seize bits, trois signaux de sélection du champ mémoire utilisé (programme, données ou entrées/sorties), et des signaux de contrôle.
Voici un exemple d'implémentation du DSP en utilisation monoprocesseur :

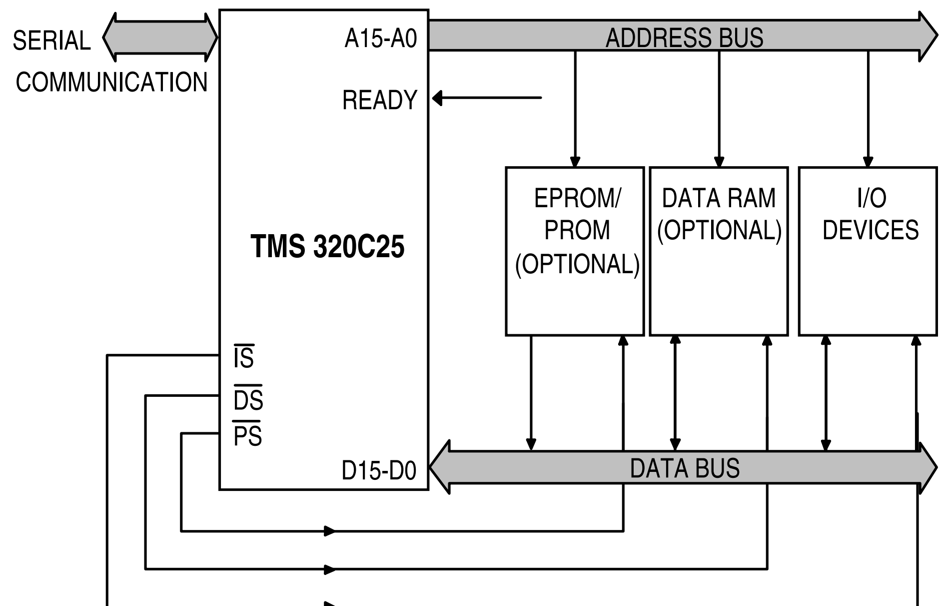


Figure 1 : le TMS 320C25 en configuration monoprocesseur.

Une RAM et une PROM externes ont été rajoutées au système minimum. Le signal READY permet de générer des états d'attente pour se synchroniser avec les mémoires externes, ou des périphériques lents. L'interface de communication permet d'adapter un CODEC, un convertisseur analogique/numérique ou un autre système minimum pour aboutir à une configuration multiprocesseur.

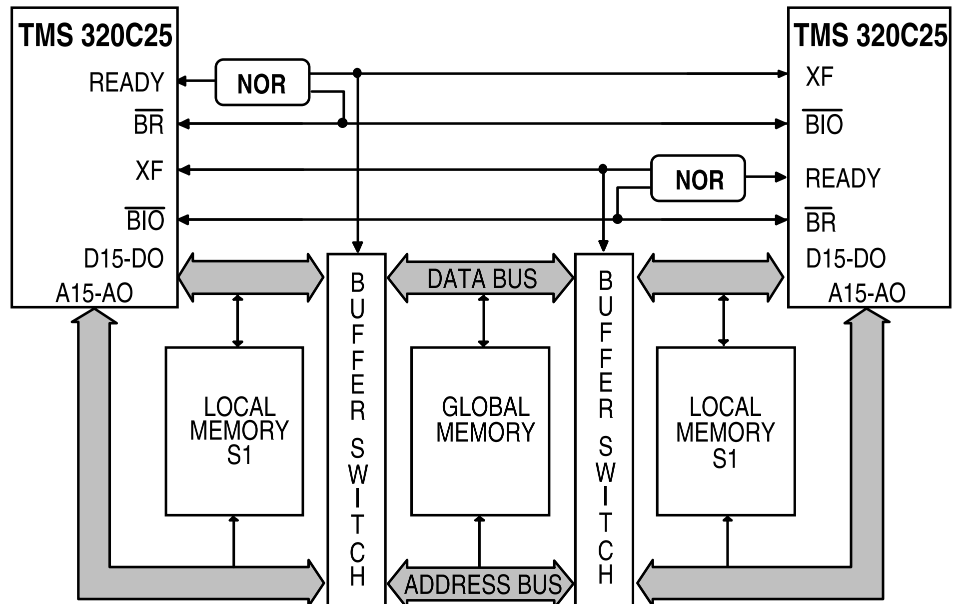
Configuration multiprocesseur

Pour ce type d'application, le TMS 320C25 permet d'adresser une mémoire commune et de gérer son partage par les signaux BR (*Bus Request*) et READY.
Le registre 8 bits interne d'adressage de la mémoire commune permet d'affecter jusqu'à 32K mots de la mémoire de données à la mémoire commune. Si l'instruction à exécuter doit adresser une opérande dans la mémoire commune, alors une demande d'accès au bus (BR) est automatiquement générée. Le signal READY prolonge alors le cycle en cours en incorporant des états d'attente jusqu'à ce qu'il change d'état autorisant ainsi le DSP à accéder à la mémoire.
Dans une telle configuration, le champ d'adresse de chacun des processeurs est divisé en deux parties :

- ❑ la mémoire locale accessible uniquement par le DSP auquel elle est affectée,
- ❑ la mémoire commune accessible par tous les DSP.

La figure suivante montre une configuration en parallèle mettant en commun une partie de la mémoire.

Figure 2 : le TMS 320C25 en configuration multiprocesseur.



Les deux DSP partagent la mémoire commune alors qu'ils exécutent leur propre programme. L'arbitrage des accès à la mémoire commune est réalisé par le matériel avec les signaux XF, BR, BIO et READY. Lorsque le signal BR est activé par une demande d'adressage à la mémoire globale, le signal BIO conditionne l'instruction de branchement BIOZ de l'autre DSP, qui doit alors exécuter et libérer le bus en positionnant le drapeau correspondant à la sortie XF. Le signal READY du DSP demandeur passe alors à 1 et l'accès à la mémoire peut se faire sans conflit.
Le fonctionnement en multiprocesseur peut également être implémenté par l'utilisation des signaux d'interface HOLD et HOLDA accompagnés par des interruptions. HOLD et HOLDA (demande de suspension de l'activité du DSP et son acquittement) permettent à un autre processeur de lire ou d'écrire dans la mémoire externe. L'adressage des mémoires données et programme est alors accessible car le TMS 320C25 se trouve dans le mode HALT suspendant l'activité du processeur et libérant les bus. Ce type de fonctionnement est très intéressant pour initialiser et pour changer le programme à exécuter par le processeur.

Modes d'adressage et instructions

Le jeu d'instruction du TMS 320C25 se divise en deux catégories : les instructions propres au traitement du signal, et les instructions de contrôle du processeur.

Pour obtenir une vitesse d'exécution optimale, une instruction est exécutée alors que la suivante est décodée, selon le mécanisme du *pipeline*.
Trois modes d'adressage sont mis en oeuvre dans ce processeur :
- mode direct,
- mode indirect,
- mode immédiat.

Ce mécanisme est connu sous le nom de mémoire segmentée, bien connue des utilisateurs des processeurs d'intel 80X86.

Les deux premiers sont utilisables pour l'adressage de la mémoire de données. Lorsque l'adressage direct est utilisé, l'adresse 16 bits est séparée en deux pour laisser un champ de 9 bits au pointeur de page de la mémoire données. Les 7 bits restant permettent d'adresser 128 mots constituant la page pointée par `DP`. Le `DP` peut adresser 512 pages de 128 mots et autorise le DSP à gérer directement 64 K mots de mémoire de données.

Les 7 bits d'adresse compris dans l'instruction adressent une case mémoire de la page de 128 mots sélectionnés par `DP`. L'adressage direct peut être utilisé avec toutes les instructions excepté avec le CALL, les instructions de branchement et les instructions utilisant l'adressage immédiat ou les instructions sans opérandes.

Le mode d'adressage indirect est aussi puissant que souple du fait de l'utilisation des 8 registres auxiliaires prévus à cet effet. Le mode d'adressage indirect avec inversion de bit réorganise les données mémoire pour l'implémentation directe d'une `FFT`.

Fast Fourrier Transform.

Résumé

Le DSP TMS320C25 permet de concevoir des configurations multiprocesseurs en offrant tous les mécanismes de contrôle d'accès aux ressources partagées. Il permet de plus de dissocier mémoire locale et mémoire commune de part ses modes d'adressages.

Le jeu d'instruction orienté vers le traitement du signal de ce processeur est bien développé, permettant par exemple une réorganisation de la mémoire en vue d'effectuer une `FFT`.

L'établissement d'une mémoire segmentée autorise l'implémentation de procédures complexes mettant en oeuvre des mécanismes évolués pour contrôler l'exécution d'un programme.

La mise en parallèle des DSP TMS 302C25 est facilitée au niveau du matériel. Il est possible d'interfacer ce processeur avec d'autres processeurs d'un autre type avec une architecture maître-esclave sans grande difficulté.

Parallélisme

De nombreuses équipes de recherche ont travaillé sur ce sujet et les références bibliographiques sont assez importantes. Il existe deux manières de piloter des processeurs en parallèle : le **parallélisme asynchrone** et le **parallélisme synchrone**.

Parallélisme asynchrone

Il existe deux sortes de parallélisme asynchrone, le parallélisme data-driven, ou piloté par les données, et le parallélisme demand-driven, ou piloté par la demande. Il y a dans les deux cas des délais d'attente, le parallélisme étant déduit au moment de l'exécution.

La première met en place un certain nombre de processeurs indépendants et une zone de **ressource partagée** pour leur communication.

Chaque processeur se voit assigner une tâche ; il puise les paramètres pour réaliser cette tâche dans la zone partagée. Une fois la tâche terminée, il place ses résultats dans la ressource partagée. L'accès à cette ressource est **bloquant**, ce qui induit des états d'attente, chaque processeur attendant ses paramètres. L'expérience montre que ce parallélisme convient bien si le grain est "assez gros", autrement dit que les tâches assignées aux processeurs sont importantes. L'expérience montre aussi qu'au-delà d'un certain nombre de processeur, une grande partie du temps est passé à attendre la disponibilité des ressources. Or l'on demande à une application de traitement du signal d'être rapide, mais surtout d'être déterministe. Cela signifie que le temps de calcul doit être constant quel que soit l'environnement de travail.

L'avantage du parallélisme asynchrone est que le nombre de processeurs peut être modifié à volonté car le séquencement est déduit dynamiquement au moment de l'exécution. Le compilateur est simple et à rapprocher des compilateurs monoprocesseur. Par contre l'architecture matérielle doit mettre en place des canaux de communication bloquants. De plus, le temps nécessaire à un calcul n'est pas déterministe.

Parallélisme synchrone Dans ce contexte, il existe une autre sorte de parallélisme, le parallélisme synchrone. Ici, l'architecture matérielle n'est pas très différente de la précédente, à ceci prés que la ressource partagée n'est pas bloquante.

Le parallélisme est extrait en amont, **au moment de la compilation**. C'est le compilateur qui résout les conflits lors de l'accès à telle ou telle ressource. Ici la difficulté est reportée au niveau du compilateur, ce qui simplifie l'architecture matérielle. De plus cette méthode produit un parallélisme à grain fin.

L'avantage du parallélisme synchrone est la simplicité de l'architecture matérielle finale, toute l'ingénierie étant reportée au niveau du compilateur. La compilation s'effectue sur une architecture cible qui ne peut être modifiée par la suite sans recompilation de l'application ; dans les applications de traitement du signal, cela ne pose pas vraiment de problème. Le temps de calcul est déterministe et peut être connu au moment de la compilation, ce qui est fondamental.

Aspect logiciel

La plus simple façon de décrire une application est d'en faire un schéma. Il segmente l'application en blocs fonctionnels, ces blocs étant reliés entre eux par des traits matérialisant la circulation des données, ou les dépendances fonctionnelles. L'ambition du projet est de fournir un logiciel capable de comprendre un tel langage de boîtes. Bien sûr, pour ce faire il s'appuiera sur les interfaces graphiques des ordinateurs d'aujourd'hui.

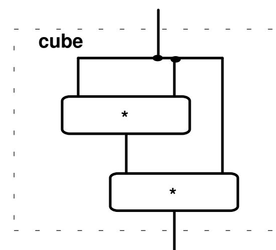
Un tel langage de boîte est parfaitement décrit par les **graphes Data Flow**. Il existe aussi des schéma de flots de contrôle.

Graphes Data Flow Un graphe Data Flow est une forme d'expression qui permet de décrire une application. On rencontre souvent de tels graphes dans les schémas électroniques où l'on trouve des blocs fonctionnels connectés entre eux.

Les graphes Data flow ont pour ambition de mimer ce comportement.

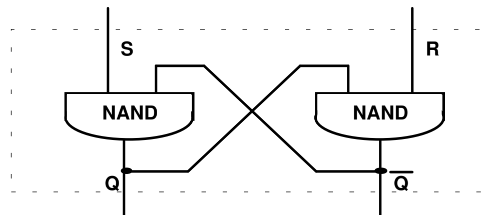
Les graphes Data flow permettent parfaitement de représenter une application acyclique comme :

Figure 3 : Graphe Data Flow de la fonction cube.



Mais un certain nombre de difficultés surgissent lorsque l'on désire par exemple introduire des boucles de contre-réaction. On pressent ces difficultés, par exemple, dans le graphe suivant :

Figure 4 : graphe Data Flow de la bascule SR.



Quelle est la valeur initiale de la boucle ? Comment empêcher une boucle de contre-réaction d'entrer dans un processus de calcul sans fin ?
Nous verrons dans la section consacrée aux graphes Data Flow qu'il existe un certain nombre de modèles théoriques. Leur fonctionnement est sensiblement différent, mais le comportement est identique, et les problèmes posés demeurent.

Langages fonctionnels

Les graphes Data Flow sont une forme d'expression immédiate. Cependant, elle ne convient pas dans tous les cas de figure. Il est donc indispensable que les graphes Data flow puissent se "réduire" en une forme plus universelle.
On pourrait à première vue penser qu'une traduction du graphe dans un langage descriptif serait une solution.
Mais nous avons entr'aperçus que les Graphes Data Flow nous réservaient un certain nombre de difficultés ; ces difficultés seront peut-être trop limitatives et cette voie abandonnée par la suite.
La démarche que nous avons entreprise dans notre équipe est toute différente. N'est-il pas concevable de trouver un langage dont un sous ensemble puisse être représenté sous la forme d'un graphe Data Flow ?
Le langage Lucid nous incite à penser que les langages fonctionnels sont les candidats privilégiés. En effet ils reposent sur une théorie dérivée des mathématiques. De ce fait ils permettent de prouver, au sens mathématique du terme, la validité d'une application, ce dont sont incapables les langages traditionnels. De plus ils décrivent une application sous forme d'un ensemble d'équations.

Nous verrons que la séquence ne disparaît pas complètement, mais quand elle est utilisée, sa portée est limitée et bien localisée.

La séquence dans de tels langages n'existe plus (la séquence imposée par l'ordre d'écriture n'a pas d'importance). Cela se traduit par une parallélisation évidentes de l'application car les équations peuvent être "évaluées" dans n'importe quel ordre sans modifier le résultat final.
Nous allons donc commencer notre étude par les langages fonctionnels et leur base théorique connue, le **λ-calcul**. Le nom du projet n'a donc pas été choisi par hasard!
Puis nous étudierons les graphes Data Flow. Nous implémenterons les différents modèles théoriques pour "mesurer" leurs performances respectives et comprendre leur comportement dans des situations critiques. Nous profiterons de cette occasion pour démontrer si besoin en était, la puissance d'expression extraordinaire des langages fonctionnels, et plus particulièrement de celui que notre équipe a choisi, le langage Scheme. En effet c'est avec celui-ci que nous avons écrit les différents modèles de graphes Data Flow.

PARTIE 1: LANGAGES FONCTIONNELS

Nous commencerons cette étude par une introduction au λ-calcul, modèle formel de tous les langages fonctionnels existants. Il s'appuie sur une théorie solide, fiable. Il implique peu de concept, ce qui le rend particulièrement adaptable et général.
Puis nous ferons un rapide tour d'horizon sur les langages proprement dit. Nous ne prétendons pas faire une analyse poussée de ces langages, mais plutôt mettre en évidence ce qui les rassemble et ce qui les sépare.
Enfin nous présenterons d'une manière plus approfondie le langage Scheme, essentiellement dans ces principes. Nous verrons qu'il est, dans une certaine mesure, un parfait candidat pour être un interpréteur de λ-expressions.

CHAPITRE 1: LE λ–CALCUL

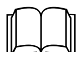
Ce chapitre présente les bases du λ-calcul et les concepts qui lui sont liés. Il est indispensable de connaître le λ-calcul avant de pouvoir étudier les langages fonctionnels. Cette démarche permet d'avoir un autre regard sur ces langages, et d'en extraire les points communs, tout comme les différences. En étudiant quelques langages fonctionnels, le lecteur ne pourra pas s'empêcher de penser à ce chapitre essentiel.

Ici, nous abordons seulement les bases du λ-calcul, et nous avons passé sous silence par exemples les combinateurs de réduction des graphes.

Cette introduction au λ-calcul s'inspire fortement du livre de SL. Peyton Jones[19]. Nous avons choisi cette démarche conscients que ce livre fait une présentation lumineuse du λ-calcul.

Un autre ouvrage plus technique est à recommander au lecteur soucieux d'approfondir ses connaissances en λ-calcul est le livre de G.Revesz[12]

Qu'est que le λ-calcul



Le λ-calcul est une théorie universelle qui décrie la notion de fonction mathématique. Les créateurs sont A.Chruch et HB Curry qui ont commencé leurs travaux dans les années trente.

C'est un formalisme de description dont la syntaxe et la sémantique sont extrêmement simples. Il est aussi suffisamment expressif pour que la plupart des langages fonctionnels s'en servent comme langage intermédiaire.

Lisp est dans sa forme originale fortement inspiré par le λ-calcul. Le passage de paramètres par nom (*call by name*) utilisé dans les langages dérivés d'Algol s'appuie sur les mécanismes de substitution formelle telle que définie précisément dans le λ-calcul.

Landin[5] reprit les travaux de Church dans un article important intitulé les 700 prochains langages ; en effet, il avait bien vu que les langages de la prochaine génération, fonctionnels, s'appuieraient tous à des degrés différents sur les concepts décrits par le λ-calcul.

Ce formalisme jouera aussi un rôle important dans la théorie des algorithmes parallèles. En effet, c'est une abstraction mathématique de l'ordonnancement, au même titre que la machine de Turing. Mais, alors que la machine de Turing est conceptuellement séquentielle, le λ-calcul met en évidence le parallélisme que l'on trouve naturellement dans les expressions mathématiques.

Les langages utilisent généralement la notation infixée : opérande opérateur opérande.

Le λ-calcul utilise la notation préfixée. Voici par exemple une λ-expression :

```
(+ 4 5)
```

En utilisant les parenthèses dans une expression plus complexe, on aurait:

```
(+ (* 5 6)(* 8 3))
```

C'est expressions sont évaluées par l'Interpréteur. Cette évaluation effectue une **réduction** de l'expression jusqu'à ce qu'elle ne soit plus réductible. Le résultat est alors affiché. Les expressions réductibles sont appelées **radicaux**. Dans le dernier exemple, il y a deux radicaux : `(* 5 6)` et `(* 8 3)`. L'expression entière `(+ (* 5 6)(* 8 3))` n'est pas un radical car l'opérateur + doit être appliqué à deux nombres avant d'être réductible. En choisissant arbitrairement le premier radical, nous obtenons:

```
(+ (* 5 6)(* 8 3)) ⇒ (+ 30 (* 8 3)),
```

Il existe d'autres formes de réduction dont nous parlerons plus loin.

où le symbole ⇒ se lit "*se réduit en*". Il ne reste qu'un radical, `(* 8 3)`, ce qui donne :

```
(+ 30 24)
```

Cette réduction a créé un nouveau radical, qui se réduit en :

```
(+ 30 24) ⇒ 54
```

Lorsqu'il n'y a plus de radicaux dans l'expression en cours d'évaluation le résultat est affiché.

Lorsqu'il y plusieurs radicaux comme dans le cas précédent, nous avons à faire le choix de celui qui sera réduit en premier. Nous pouvons déjà remarquer que le parallélisme implicite des équations mathématiques apparaît immédiatement. Ce point sera abordé plus loin.

Application et curryfication

Dans le λ-calcul, l'application de fonction est si fréquente qu'elle est notée par une simple juxtaposition; nous écrivons :

```
f x
```

pour indiquer "*la fonction f appliquée à l'argument x*". Comment peut-on exprimer l'application d'une fonction à plusieurs arguments ? Nous utiliserons la notation :

```
((f x) y)
```

Par exemple :

```
((+ 3) 4)
```

Le λ-calcul permet à une fonction de rendre une fonction (comme dans la plupart des langages fonctionnels)

L'expression `(+ 3)` représente la "*fonction qui ajoute 3 à son argument*". L'expression entière est donc "*la fonction + appliquée à 3, dont le résultat est une fonction que l'on applique à 4*".

Ce mécanisme nous permet donc de voir toutes les fonctions comme prenant un seul argument. Il a été introduit par Schönfinkel en 1924 et utilisé intensivement par Curry et Feys en 1958 ; ce mécanisme est connu sous le nom de **curryfication**

Parenthèses

En mathématiques, il est courant d'omettre les parenthèses superflues pour ne pas surcharger les expressions. Par exemple, nous pourrions omettre les parenthèses dans l'expression :

```
(ab)+(((2c)/d)
```

ce qui donnerait :

```
ab + 2c / d
```

La seconde expression est plus lisible que la première mais elle court le risque d'être ambiguë. L'établissement de convention de précédence entre les différentes fonctions permet de lever ces ambiguïtés.

Par exemple, la multiplication a une précédence plus forte que l'addition.

Quelquefois, on ne peut omettre certaines parenthèses, comme dans l'expression :

```
(b+c)/a
```

Des conventions similaires sont utiles pour l'écriture d'expressions du λ-calcul. Considérons l'expression :

```
((+ 3) 2)
```

En établissant la convention que **l'application de fonction associe à gauche**, nous pouvons écrire cette expression plus simplement :

```
(+ 3 2)
```

ou même :

Nous présentons ici une syntaxe du λ-calcul simplifiée.

```
+ 3 2
```

Nous avons déjà utilisé de telles conventions dans les exemples précédents. Fournissant un exemple plus complexe, l'expression :

```
((f(+ 4 3))(g x))
```

est complètement parenthésée et non ambiguë. En suivant notre convention, nous pouvons omettre les parenthèses superflues, rendant ainsi l'expression plus lisible, ce qui donne :

```
f(+ 4 3) (g x)
```

Aucune parenthèse supplémentaire ne peut être omise. Il peut, bien sûr, en être ajoutées librement sans changer le sens de l'expression ; par exemple :

```
(f (+ 4 3) (g x))
```

λ-abstraction

Le λ-calcul possède une construction, appelée **λ-abstraction**, pour représenter de nouvelles fonctions. C'est le moyen d'extension du langage. Une λ-abstraction est une espèce particulière d'expression représentant une fonction. Voici un exemple de λ-abstraction :

```
(λx.+ x 1)
```

Le λ signifie "*voici une fonction*", et est immédiatement suivi par une variable, x en l'occurrence ; ensuite vient un "." suivi par le **corps de la fonction**, `(+ x 1)` dans le cas présent. La variable est appelée **paramètre formel**, et on dit que le λ le **lie**. On peut lire l'expression de la manière suivante :

```
( λ               x   .   +       x   1
  ↑               ↑   ↑   ↑       ↑   ↑
La fonction de    x   qui ajoute  x   et 1
```

Une λ-abstraction comporte **toujours** ces quatre composants : le λ, le paramètre formel, le "." et le corps.

Une λ-abstraction est similaire à une définition de fonction dans un langage conventionnel tel que le **C** :

```
int Inc (x) int x; {
  return (x + 1);
}
```

Le paramètre formel de la λ abstraction correspond au paramètre formel de la fonction, et le corps de l'abstraction est une expression plutôt qu'une suite de commandes.
Cependant, les fonctions d'un langage conventionnel doivent posséder un nom (*Inc* par exemple), alors que les λ-abstractions sont des **fonctions anonymes**.

La sémantique du λ-calcul

Nous n'avons décrit jusqu'à présent que la syntaxe du λ-calcul, nous devons maintenant justifier son titre de "*calcul*". A cet effet, nous donnerons trois règles de conversion décrivant comment convertir une λ-expression en une autre.
Introduisons d'abord une terminologie importante.

Variables libres et liées

Considérons la λ-expression:

`(λx. + x y) 4`

Dans le but d'évaluer complètement cette expression, nous devons connaître la valeur "*globale*" de `y`. Par contre, nous n'avons pas besoin de connaître une éventuelle valeur "*globale*" de `x`, cette variable étant le paramètre formel de la fonction. Nous voyons donc que `x` et `y` ont des statuts différents.

La raison en est que `x` apparaît **liée** par le `λx` ; il s'agit donc seulement de la position à laquelle viendra se placer l'argument `4` lors de l'application de la λ-abstraction à cet argument.

Nous verrons par la suite qu'une des difficultés d'implémentation est la gestion des variables libres.

Par contre, `y` n'étant liée par aucun λ apparaît **libre** dans l'expression. En règle générale, la valeur d'une expression ne dépend que de la valeur de ses variables libres.

β-conversion

Une λ-abstraction représente une fonction, nous devons donc décrire comment l'appliquer à un argument. Par exemple :

`(λx.+ x 1)4`

est la juxtaposition de la λ-abstraction `(λx.+ x 1)` et de l'argument `4`. La règle pour une telle application de fonction est très simple :

Nous utilisons les termes "instance", "instancier" et "instanciation" comme des anglicismes signifiant respectivement "cas particulier", (ou "représentant"), "particulariser" et "particularisation"

le résultat de l'application d'une λ-abstraction à un argument est une instance du corps de la λ-abstraction dans laquelle les occurrences (libres) du paramètre formel dans le corps sont remplacées par des copies de l'argument.

Donc le résultat de l'application de la λ-abstraction `(λx.+ x 1)` à l'argument `4` est :

`+ 4 1`

L'expression `(+ 4 1)` est une instance du corps `(+ x 1)` dans laquelle les occurrences du paramètre `x` sont remplacées par l'argument `4`. Nous écrivons la réduction en utilisant la flèche "→" comme auparavant :

`(λx.+ x 1)4 → + 4 1`

Cette opération est appelée **β-réduction**. Nous utiliserons quelques exemples afin de détailler le fonctionnement de la β-réduction.

Exemples de β-réductions

Le paramètre formel peut apparaître plusieurs fois dans le corps :

```
(λx.+ x x) 5 → + 5 5
             → 10
```

Il peut aussi n'y avoir aucune occurrence du paramètre formel dans le corps :

Dans ce cas il n'existe pas d'occurrence du paramètre formel (x) auquel l'argument (5) puisse être substitué, et donc l'argument est inutilisé.

```
(λx.3) 5 → 3
```

Le corps d'une λ-abstraction peut être une autre λ-abstraction :

```
(λx.(λy.- x y))4 5    → (λy.- y 4)5
                      → - 5 4
                      → 1
```

Nous abrégeons souvent (λx.(λy.E)) en (λx.λy.E)

Remarquons que lors de la construction d'une instance du corps de l'abstraction λx, nous copions le corps entier, incluant l'abstraction λy. Nous voyons ici la curryfication : l'application de l'abstraction λx renvoie une fonction en résultat, qui, lors de son application, retourne le résultat (-5 4).

Les fonctions peuvent aussi être utilisées comme arguments :

```
(λf.f 3)(λx.+ x 1)  → (λx.+ x 1)3
                    → + 3 1
                    → 4
```

Une instance de l'abstraction λx est substituée à f là où f apparaît dans le corps de l'abstraction λf.

Noms de variables

La prudence est de rigueur lorsque les noms des paramètres formels ne sont pas uniques. Par exemple :

```
(λx.(λx.+(- x 1))x 3)9     → (λx.+ x 1)9 3
                           → + (- 9 1)3
                           → 11
```

On peut remarquer que nous n'avons rien substitué à l'occurrence interne de x lors de la première substitution : celle-ci était située sous un λx qui la protégeait ; autrement dit, l'occurrence interne de x n'est pas libre dans le corps de l'abstraction externe λy.

Etant donné une abstraction (λx.E), comment peut-on identifier exactement les occurrences de x concernées par la substitution ? Nous devrons simplement substituer à toutes les occurrences de x qui sont libres dans E, car, si elles sont libres dans E, elles seront liées par λx dans (λx.E). En appliquant l'abstraction λ x la plus externe, après examen de son corps :

```
(λx.+ (- x 1)) x 3
```

Nous remarquons que seules la deuxième occurrence de x est libre et donc substituable.

L'emboîtement des portées de variables dans les langages structurés est analogue

C'est pourquoi la règle donnée précédemment spécifiait que seules les occurrences **libres** du paramètre formel dans le corps sont candidates à la substitution.

Un exemple

Nous allons montrer comment les constructeurs de données peuvent être modélisés par des λ-abstractions pures.

Nous définissons CONS, HEAD et TAIL par :

Ces constructeurs de données sont à rapprocher de CONS CAR et CDR de LISP

```
CONS = (λa.λb.λf.f a b)
HEAD = (λc.c (λa.λb.a))
TAIL = (λc.c (λa.λb.b))
```

Le lecteur est invité à faire d'autres démonstrations par l'exemple.

Montrons sur un exemple que ces fonctions agissent comme nous l'attendons :

```
HEAD (CONS p q))    = (λc.c (λa.λb.a))(CONS p q)
                    → (CONS p q)(λa.λb.a)
                    = ((λa.λb.λf.f a b) p q)(λa.λb.a)
                    → ((λb.λf.f p b) q)(λa.λb.a)
                    → (λf.f p q)(λa.λb.a)
                    → (λa.λb.a) p q
                    → p
```

Cela est satisfaisant d'un point de vue théorique, mais toutes les implémentations possèdent des fonctions pré définies pour des raisons d'efficacité.

Ce qui signifie, en particulier, que les fonctions pré-définies `CONS`, `HEAD` et `TAIL` ne sont pas essentielles. Il s'avère que toutes les autres fonctions pré-définies peuvent être écrites à l'aide de λ-expressions.

Conversion, réduction et abstraction

Nous pouvons utiliser la règle β à l'envers, pour construire de nouvelles abstractions, comme :

```
+ 4 1 ← (λx.+ x 1) 4
```

Cette opération est appelée **β-abstraction**, que nous notons à l'aide d'une flèche de réduction inversée. Une **β-conversion** est soit une β-réduction soit une β-abstraction, et nous la notons par une double flèche $\leftrightarrow_\beta$. Nous écrivons donc :

La flèche est annotée β pour distinguer les différentes réductions

$+\ 4\ 1 \leftrightarrow_\beta (\lambda x.+\ x\ 1)\ 4$

La β-conversion peut être vue comme exprimant une équivalence de deux expressions "*qui semblent différentes*" mais "*qui ont la même signification*". Nous avons besoin de deux règles supplémentaires pour satisfaire notre intuition au sujet de l'équivalence de deux expressions.

α-conversion

Considérons ces deux abstractions :

```
(λx.+ x 1)
```

et

```
(λy. + y 1)
```

Elles "*doivent*" être équivalentes. Cela est assuré par la **α-conversion** qui permet de changer le nom des paramètres formels d'une λ-expression, pourvue que ce soit réalisé de façon cohérente. Ainsi :

$(\lambda x.+\ x\ 1) \leftrightarrow_\alpha (\lambda y.\ +\ y\ 1)$

La λ-conversion est utilisée pour éviter des conflits de noms.

Le nom de la variable introduite ne doit pas apparaître libre dans le corps de la λ-abstraction originale.

η-conversion

Une règle supplémentaire de conversion est nécessaire pour exprimer nos intuitions à propos du fait que deux λ-abstractions "*doivent*" être équivalentes. Considérons les deux expressions :

```
(λx.+ 1 x)
```

et :

```
(+ 1)
```

Ces expressions se comportent exactement de la même manière lorsqu'elles sont appliquées à un argument : elles lui ajoutent 1. La **η-conversion** est une règle exprimant leur équivalence :

`(λx.+ x 1)` $\leftrightarrow_\eta$ `(+ 1)`

Plus généralement, nous pouvons exprimer la règle de η-conversion comme suit :

`(λx.F x)` $\leftrightarrow_\eta$ `F`

si `x` n'apparaît pas dans `F`, et `F` représente une fonction. La condition de liberté empêche de réaliser de fausses conversions. Par exemple :

`(λx.+ x x)`

n'est pas η-convertible avec :

`(+ x)`

car `x` apparaît libre dans `(+ x)`. La condition imposant à `F` de représenter une fonction empêche de réaliser de fausses conversions de constantes pré-définies ; par exemple :

`TRUE`

n'est pas η-convertible avec :

`(λx.TRUE x)`

Lorsqu'elle est utilisée de gauche à droite, la η-conversion est appelée η-réduction.

Capture des noms de variables

Nous attirons l'attention du lecteur sur le problème de la capture des noms de variables en donnant un exemple montrant que le λ-calcul est plus épineux qu'il n'y paraît.

Supposons définir une λ-abstraction par :

Notation simplifiée.

`TWICE = (λf.λx.f (f x))`

Considérons maintenant la réduction de l'expression `(TWICE TWICE)` par β-réduction :

```
TWICE TWICE
= (λf.λx.f (f x)) TWICE
→ (λx.TWICE (TWICE x))
```

Maintenant, il y a deux radicaux β, `(TWICE x)` et `(TWICE (TWICE x))`. Choisissons (arbitrairement) de réduire le plus interne, en remplaçant d'abord `TWICE` par sa λ-expression :

```
→ (λx.TWICE ((λf.λx.f (f x))x))
```

Nous voyons maintenant le problème. Pour appliquer `TWICE` à `x`, nous devons fabriquer une nouvelle instance de corps de `TWICE` (souligné) en remplaçant les occurrences du paramètre formel `f` par l'argument `x`.

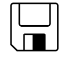
Mais `x` est **déjà utilisé comme paramètre formel** à l'intérieur du corps de l'abstraction. Il est clairement faux de réduire en :

```
(λx.TWICE ((λf.λx.f (f x))x))
→ (λx.TWICE (λx.x (x x)))
```

car dans ce cas, le x substitué à f serait "*capturé*" par la λ-abstraction la plus interne. On appelle cela le problème de la **capture des noms de variables**. Une solution est d'utiliser la α-conversion afin de changer le nom de l'un des λx ; par exemple :

```
(λx.TWICE ((λf.λx.f (f x))x))
↔α (λx.TWICE ((λf.λy.f (f y))x))
↔α (λx.TWICE (λy.x (x y)))
```

Nous concluons que:



- La β-réduction n'est valide que si les occurrences libres de variables de l'argument n'entrent en conflit avec aucun paramètre formel du corps de la λ-abstraction
- la α-conversion est quelquefois nécessaire afin d'éviter la condition précédente

δ-conversion

On peut considérer les fonctions pré-définies comme une forme supplémentaire de conversion : la **δ-conversion**. Pour cette raison, les règles de réduction relatives aux fonctions pré-définies sont quelquefois appelées **δ-règle**s.
Comme nous l'avons vu, l'application de ces règles de conversion n'est pas toujours immédiate, ce qui nous amène à en donner une définition formelle. A cet effet, une notation supplémentaire nous sera utile.
La notation E[M/x] représente l'expression E dans laquelle M est substitué à toutes les occurrences libres de x.
On peut lire E[M/x] par : "*E dans laquelle M remplace les x*", de telle sorte que x[M/x] = M. Cette notation nous permet d'exprimer simplement la β-conversion par :

```
(λx.E) M ↔β E[M/x]
```

et est utilisé pour exprimer aussi la α-conversion.

Résumé

Nous avons introduit trois règles de conversion qui nous permettent inter-convertir des expressions contenant des λ-abstractions. Il s'agit :

- du **changement de noms** : la α-conversion nous permet de changer le nom du paramètre formel d'une λ-abstraction, en prenant quelques précautions.
- de l'application de fonction : la β-réduction permet d'**appliquer une λ-abstraction** à un argument par substitution, en prenant quelques précautions (variables libres).
- de l'**élimination des λ-abstractions superflues** : la η-réduction peut quelque fois éliminer une λ-abstraction.

Ordre de réduction

Si une expression ne contient aucun radical, alors l'évaluation est terminée et l'expression est dite en **forme normale**.
Cependant, une expression peur contenir plus d'un radical et, dans ce cas, la réduction peut s'exécuter selon plusieurs chemins. Par exemple, l'expression (+ (* 3 4)(* 7 8)) peut être réduite en forme normale par :

```
(+ (* 3 4)(* 7 8)) → (+ 12 (*7 8))
                   → (+ 12 56)
                   → 56
```

ou par :

```
(+ (* 3 4)(* 7 8)) → (+ (* 3 4) 56)
                   → (+ 12 56)
                   → 56
```

Toute expression ne possède pas une forme normale ; considérons par exemple :

```
(D D)
```

où `D` est `(λx.x x)`. L'évaluation de cette expression ne termine pas car `(D D)` de réduit en `(D D)` :

```
(λx.x x)(λx.x x)  → (λx.x x)(λx.x x)
                  → (λx.x x)(λx.x x)
```

Une boucle infinie, par exemple.

Cette situation correspond au cas d'un programme impératif qui ne se termine pas.
De plus, **certaines** suites de réduction peuvent aboutir à une forme normale, alors que d'**autres ne se terminent pas**. Par exemple :

```
(λx.3)(D D)
```

Si nous réduisons d'abord l'application de `(λx.3)` à `(D D)` (sans réduire `(D D)`), nous obtenons le résultat `3`, mais si nous choisissons de réduire `(D D)`, alors la réduction ne se terminera pas.

Réduction en ordre normal

Ces problèmes soulèvent une question embarrassante : deux chemins différents de réduction peuvent-ils aboutir à des formes normales différentes ? Il est indispensable que la réponse soit négative.

Heureusement, elle l'est : c'est la conséquence de deux théorèmes puissants : les **théorèmes de Church-Rosser I et II**.

> THEOREME
>
> **Théorème I de Church-Rosser (CRT1)**
> si $E_1 \leftrightarrow E_1$, alors il existe une expression E telle que :
> $E_1 \rightarrow E$ et $E_2 \rightarrow E$

Le corollaire suivant est une conséquence immédiate :

Informellement, ce corollaire affirme que toutes les suites de réductions qui terminent aboutirons sur le même résultat

- **Corollaire** : aucune expression ne peut être convertie en deux formes normales distinctes (i.e. qui ne sont pas α-convertibles).
- **Preuve** : supposons que $E \leftrightarrow E_1$ et que $E \leftrightarrow E_2$, où E_1 et E_2 sont en forme normale. Alors, $E_1 \leftrightarrow E_2$ et, par CRT1, il doit exister une expression F telle que $E_1 \rightarrow F$ et $E_2 \rightarrow F$. Mais E_1 et E_2 ne contiennent pas de radicaux, donc $E_1 = F = E_2$.

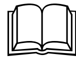
Le second théorème de Church-Rosser concerne un ordre particulier de réduction appelé l'**ordre normal**.

THEOREME

Théorème II de Church-Rosser (CRT 2)
si $E_1 \rightarrow E_2$, et si E_2 est en forme normale, alors il existe une suite de réductions de E_1 vers E_2 suivant l'**ordre normal.**

Ce théorème indique qu'il y a au plus un résultat, et l'ordre normal de réduction y aboutira si ce résultat existe. Remarquons aussi qu'aucune suite de réduction ne peut aboutir à un résultat incorrect, le pire qui puisse arriver est la non-terminaison.

La réduction en ordre normal spécifie que le radical <u>le plus extérieur et le plus à gauche</u> doit être réduit en premier.



Les preuves les plus concises du théorème I de Church-Rosser (qui est le plus difficile) se trouvent dans [Welch 75] et [ROSSER 82].

Ordre de réduction optimal

L'ordre normal de réduction garantit de trouver une forme normale (si elle existe), mais ne garantit pas de la trouver par un nombre minimal de réduction. En fait, pour la réduction d'arbres, on peut montrer que c'est la moins favorable, mais il semble que si on considère la réduction des graphes, l'ordre normal est presque optimal, et il est probablement plus coûteux de trouver le radical préservant l'optimalité que de suivre l'ordre normal.

Le problème de trouver des ordres de réductions pratiquement optimaux préservant les bonnes propriétés de l'ordre normal a été traité dans [LEVY 80].

Pour la réduction d'expressions de combinateurs `S` et `K`, l'ordre normal de réduction de graphes a été prouvé optimal. Ce résultat, parmi beaucoup d'autres concernant la réduction des graphes est prouvé dans la série de communications de Staples [STAPLE 80a, 80b, 80c]. Un traitement plus accessible de ce travail est donné par [KENNAWAY 84].

Fonctions récursives

Nous avons annoncé la possibilité de traduire tous les langages fonctionnels en λ-calcul. La récursion est une caractéristique très utilisée dans les programmes fonctionnels, or le λ-calcul ne possède pas de construction ressemblant à la récursion.

Ceci montre, si besoin en était, la puissance expressive du λ-calcul.

Dans les sections qui vont suivre, nous montrerons que le λ-calcul est capable d'exprimer des fonctions récursives sans aucune extension.

Notons tout de même que dans le cadre de notre application, à savoir la programmation multi-DSP, qu'il est probable que nous soyons dans l'incapacité de programmer des fonctions récursives, et ce pour plusieurs raisons. D'une part ce type de fonction n'est pas déterministe, et d'autre part, les processeurs DSP ne possèdent pas de pile, élément essentiel dans le processus récursif.

Fonction récursive Y

Considérons la définition récursive de la fonction factorielle :

```
FAC = (λn.IF (=n 0)1(* n (FAC (- n 1))))
```

Cette définition repose sur le fait de pouvoir nommer une λ.abstraction et de faire référence à ce nom dans la λ-abstraction elle-même. Il n'y a pas de telle construction

dans le λ-calcul. Le problème étant que les λ-abstractions sont des fonctions anonymes, elles ne peuvent pas se nommer (et donc faire référence à) elle-même.

Nous examinerons ce problème dans le cas où la récursion est réduite à sa forme la plus simple. Commençons par une définition récursive :

Nous avons noté certaines parties du corps "..." afin de mieux porter notre attention sur la récursivité.

```
FAC = λn.(...FAC...)    ①
```

Nous pourrions écrire cette définition sous la forme :

```
FAC = H FAC
```

où :

```
H = (λfac.(λn.(...fac...)))FAC
```

La définition de `H` est simple. C'est une λ-abstraction ordinaire et n'utilise pas de récursion. La récursion est exprimée uniquement dans la définition ① ci-dessus.

La définition ① ressemble à une équation mathématique. Par exemple, afin de résoudre l'équation mathématique :

$$x^2 - 2 = x$$

nous cherchons des valeurs de `x` qui satisfont l'équation. De la même manière, pour résoudre ①, nous cherchons pour `FAC` une λ-expression qui satisfait ①. Comme pour les équations mathématiques, il peut exister plusieurs solutions.

L'équation ① :

```
FAC = H FAC
```

spécifie que le résultat de l'application de la fonction `H` à `FAC` est `FAC` elle-même. Nous disons alors que `FAC` est un **point fixe** de `H`. Une fonction peut avoir plus d'un point fixe.

Par exemple, 0 et 1 sont les points fixes de la fonction λx. x x qui élève son argument au carré.*

Nous cherchons donc un point fixe de `H`. Clairement, cela ne peut dépendre que de `H` ; inventons donc (momentanément) une fonction `Y` qui à une fonction associe un point fixe de cette fonction. `Y` a donc le comportement suivant :

```
Y H = H (Y H)
```

et est appelée un **combinateur de point fixe**. Si maintenant nous pouvons obtenir un tel `Y`, notre problème est résolu. Nous pourrions alors donner une solution à ①, obtenant :

```
FAC = Y H
```

c'est à dire une définition non récursive de `FAC`. Afin de nous convaincre que `FAC` possède le comportement attendu, calculons `(FAC 1)`. Rappelons les définitions de `FAC` et de `H` :

```
FAC = Y H

H   = λfac.λn.IF (= n 0) 1 (* n (fac (- n 1)))
```

Et donc :

```
FAC 1
= Y H 1
= H(Y H)1
= (λfac.λn.IF (= n 0) 1 (* n (fac (- n 1))))(Y H) 1
→ (λn.IF (= n 0) 1 (* n (Y H (- n 1)))) 1
→ IF (= 1 0) 1 (* 1 (Y H (- 1 1)))
→ * 1 (Y H 0)
→ * 1 (H (Y H) 0)
→ * 1 ((λfac.λn.IF (= n 0) 1 (* n (fac (- n 1))))(Y H)0)
→ * 1 ((λn.IF (= n 0) 1 (* n (Y H (- n 1)))) 0)
```

```
→ * 1 (IF (= 0 0) 1 (* 0 (Y H (- 0 1))))
→ * 1 1
→ 1
```

Y est une λ-expression

Le fait que Y puisse être implémenté comme une λ-abstraction est satisfaisant d'un point de vue théorique mais inefficace. La plupart des implémentations fournissent Y.

Nous avons montré comment transformer une définition récursive de `FAC` en une définition non récursive, mais en utilisant une fonction nouvelle `Y`. Cette fonction doit satisfaire la propriété suivant :

```
Y H = H (Y H)
```

qui semble exprimer la récursivité dans sa forme la plus pure, car nous pouvons l'utiliser pour exprimer toutes les autres fonctions récursives. Nous pouvons définir `Y` comme une λ-abstraction sans utiliser la récursion:

```
Y = (λh.(λx.h (x x))(λx.h (x x)))
```

Afin de montrer que `Y` satisfait la propriété attendue, évaluons :

```
Y H
= (λh.(λx.h (x x))(λx.h (x x))) H
↔ (λx.H (x x))(λx.H (x x))
↔ H (Y H)
```

produisant le résultat attendu.

Sémantique dénotationnelle du λ-calcul

Il y a deux manières de regarder une fonction. On peut la voir comme un algorithme qui, étant donné un argument, produira une valeur, ou bien comme un ensemble ordonné de couples argument-valeur.

La première est une vision "*dynamique*" ou **opérationnelle**, car on voit la fonction comme une suite d'opérations dans le temps. La seconde est "*statique*" ou **dénotationnelle** : la fonction est vue comme un ensemble fixe d'associations entre arguments et valeurs correspondantes.

Dans les sections précédentes, nous avons montré comment une expression pouvait être évaluée par l'application répétée des règles de réduction. Ces règles prescrivent des transformations purement **syntaxiques** sur les expressions permises, sans faire référence au "*sens*" de l'expression ; le λ-calcul peut vraiment être vu comme un système formel de manipulation de symboles syntaxiques. Néanmoins, le développement des règles de conversion était basé sur nos intuitions au sujet de fonctions abstraites, et cela nous a effectivement fourni une **sémantique opérationnelle** du λ-calcul. Mais pourquoi supposons-nous que le λ-calcul est le moyen d'expression adéquat du concept de fonction abstraite ?

Afin de répondre à cette question, il est nécessaire de donner une sémantique dénotationnelle du λ-calcul.

La fonction Eval

Le but de la sémantique dénotationnelle d'un langage est d'attribuer une valeur à chaque **expression** du langage. Une expression est un objet syntaxique construit selon les règles syntaxiques du langage. Par contre, une valeur est un objet mathématique **abstrait**, tel que le "*nombre 7*" ou "*la fonction qui élève son argument au carré*".
Nous pouvons donc exprimer la sémantique d'un langage par une fonction (mathématique), `Eval`, des expressions vers les valeurs :

Nous pouvons maintenant écrire des équations telles que :

Les crochets indiquent que le contenu est un objet syntaxique. Cette convention est très souvent utilisée en sémantique dénotationnelle.

`Eval[+ 3 4] = 7`

La signification en est : "*le sens (i.e. la valeur) de l'expression* `(+ 3 4)` *est la valeur numérique abstraite 7*". On peut voir l'expression `(+ 3 4)` comme la représentation ou la dénotation de la valeur `7` (d'où le terme **sémantique dénotationnelle**).
Nous donnons maintenant une description informelle de la fonction `Eval` pour la λ-calcul. Cela consiste à donner une valeur à `Eval[E]`, pour toute λ-expressions `E`, et nous procédons en faisant directement référence à la syntaxe des λ-expressions.
Nous omettrons dans un premier temps la question des constantes et des fonctions pré-définies.
Supposons que `E` soit une variable `x`. Quelle serait la valeur de :

`Eval[x]`

où `x` est une variable ? Malheureusement, la valeur d'une variable est donnée par le contexte qui l'enveloppe ; nous ne pouvons donc dire quelle est sa valeur sans information supplémentaire. Nous résolvons ce problème en donnant à **`Eval`** un paramètre supplémentaire ρ, qui donne cette information contextuelle.

L'argument ρ est appelé un environnement et représente une fonction faisant correspondre une valeur à un nom de variable. Donc :

`Eval[x] ρ = ρ x`

La notation `(ρ x)` du membre droit signifie "*la fonction appliquée à l'argument x*".
Considérons les applications. Il semble raisonnable de penser que la valeur de $(E_1 E_2)$ soit la valeur de E_1 appliquée à E_2 :

$\mathtt{Eval}[E_1 E_2]\ \rho = (\mathtt{Eval}[E_1]\ \rho)(\mathtt{Eval}[E_2]\ \rho)$

Le cas restant est celui des λ-abstractions. Quelle doit être la valeur de `(Eval[λ x.E] ρ)` ?
Certainement une fonction que nous pouvons définir en donnant sa valeur lorsqu'elle est appliquée à un argument arbitraire *a* :

Suivant notre convention concernant la curryfication, nous omettrons les parenthèses dans la suite...

`(Eval[λx.E] ρ) a`

La phrase suivante résume notre intuition au sujet des λ-abstractions :

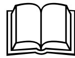
la valeur d'une λ abstraction appliquée à un argument est une valeur du corps de l'abstraction dans un contexte où le paramètre formel est lié à l'argument.
Formellement, nous écrivons :

`(Eval[λx.E] ρ) a = Eval[E] ρ[x = a]`

où la notation `ρ[x = a]` représente "*la fonction* ρ *étendue de sorte à lier la variable x à la valeur a*". Plus précisément :

`ρ[x = a] x = a`

`ρ[x = a] y = ρ y`
si `y` est une variable différente de `x`.
A l'exception des constantes et des fonctions pré-définies, nécessitant un traitement particulier, nous avons donné une sémantique dénotationnelle simple au λ-calcul.
Ici, nous avons omis une notion importante : la description de la collection de toutes les valeurs possibles produites par `Eval`. Cette collection est appelée **domaine**, et est une structure complexe qui inclut toutes les fonctions et les valeurs qui peuvent être représentées par une λ-expression. La complication provient du fait que le domaine doit inclure son propre espace de fonctions à cause de la nécessité de pouvoir appliquer une fonction à elle-même, comme dans la λ-abstraction `Y`. C'est le but de la **théorie des domaines** [SCOTT 81] que de développer une théorie fidèle de tels domaines.



Nous admettrons l'existence et la fidélité de cette théorie et de la sémantique dénotationnelle.

Notation Nous avons vu que l'environnement ρ est un argument essentiel de la fonction `Eval`. Cependant, dans toutes les utilisations futures de `Eval`, ρ ne jouera pas de rôle particulier. Ainsi, par souci de clarté, nous l'omettrons.

Le symbole ⊥

Une des caractéristiques les plus utiles de la théorie décrite dans cette section est qu'elle nous fournit la possibilité d'exprimer des raisonnements concernant la terminaison des programmes. Comme nous l'avons remarqué plus haut, la réduction d'une expression peut ne pas aboutir à une forme normale. Quelle valeur la sémantique doit-elle attribuer à de tels programmes ?

Cette valeur existe dans tous les langages fonctionnels. Elle se nomme "botom" dans Lucid, "Empty list" dans Scheme ...

Nous n'avons qu'à ajouter au domaine des valeurs possibles un élément ⊥, prononcé *bottom*, qui sera la valeur attribuée à une expression ne possédant pas de forme normale :

`Eval[expression sans forme normale] = ⊥`
L'élément ⊥ possède un sens mathématique tout-à-fait respectable dans la théorie des domaines, et, comme le symbole 0 (qui lui aussi signifie "*rien*"), son utilisation nous permet souvent d'écrire des équations condensées à la place de longues phrases.
Par exemple, au lieu de dire "l'évaluation de E ne se termine pas", nous écrirons :

`Eval[E] = ⊥`

Constantes & fonctions pré-définies

Dans cette section, nous verrons comment définir la valeur de `Eval[k]`, où `k` est une constante ou une fonction pré définie. Par exemple, quelle est la valeur de `Eval[⋆]` ? Certainement une fonction à deux arguments que nous pouvons définir en donnant sa valeur lorsqu'elle est appliquée à deux arguments arbitraires :

`Eval[⋆] a b = a x b`

Pour la multiplication, la notation et l'opération sont notés différemment. Ce n'est pas le cas, par exemple, de l'addition.

Cela donne le sens de l'opération `⋆` du λ-calcul en terme de la manipulation mathématique `x`. La distinction entre `⋆` et `x` est cruciale : `⋆` est une expression syntaxique du λ-calcul, alors que `x` est une opération mathématique abstraite.
Nous utiliserons de lettres minuscules, comme `a` et `b`, pour représenter des valeurs dans les équations sémantiques.

L'équation ci-dessus est une spécification incomplète pour $\star$. Nous devons dire comment $\star$ agit sur tous ses arguments possibles, y compris $\perp$. L'ensemble complet d'équations est en fait :

```
Eval[⋆] a b = a x b, si : a ≠ ⊥ et b ≠ ⊥
Eval[⋆] ⊥ b = ⊥
Eval[⋆] a ⊥ = ⊥
```

Ces nouvelles équations complètent la définition de $\star$ en spécifiant par exemple que si l'évaluation de l'un des deux arguments de $\star$ ne se termine pas, alors l'évaluation de l'application de $\star$ ne se terminera pas non plus.
Ce n'est pas là le seul ensemble d'équations pour un opérateur de multiplication. Par exemple, voici les équations pour un opérateur de multiplication "*plus intelligent*", $\otimes$:

```
Eval[⊗] a b = a x b, si : a ≠ ⊥, a ≠ 0 et b ≠ ⊥
Eval[⊗] 0 b = 0
Eval[⊗] a ⊥ = ⊥,      si a ≠ 0
Eval[⊗] ⊥ b = ⊥
```

Ces équations forcent $\otimes$ à évaluer son premier argument et, si celui-ci s'évalue en 0, retourne 0 sans examiner son second argument ; sinon, $\otimes$ se comporte comme $*$.

L'utilisation de $\otimes$ fait terminer l'évaluation de certaines expressions qui auraient demandé une évaluation infinie en utilisant $*$.

Cet exemple montre que les équations sémantiques d'une fonction pré définie permettent d'exprimer de subtiles variations dans son comportement, avec une précision difficile à égaler à l'aide de règles de réductions. Les équations sémantiques d'une fonction spécifient à la fois le sens de la fonction et son comportement opérationnel.
Pour être complet, nous devions donner des équations telles que :

```
Eval[6] = 6
```

où le "6" du membre gauche est une λ-expression, et celui du membre droite est un objet mathématique abstrait. Cela s'applique à toutes les constantes pré-définies, comme :

```
Eval[TRUE] = TRUE
Eval[IF]   = IF
Eval[+]    = +
```

Par exemple, utilisant cette notation plus souple, nous pourrions écrire les équations sémantiques suivantes pour la fonction `IF` :

```
IF TRUE  a b = a
IF FALSE a b = b
IF ⊥     a b = ⊥
```

Caractère strict et paresseux

Nous disons qu'une fonction est stricte si elle exige de connaître la valeur de son argument. Pouvons-nous donner une définition dénotationnelle du caractère strict ?
Si une fonction `f` a besoin de la valeur de son argument, et si l'évaluation de cet argument ne se termine pas, alors l'application de `f` à cet argument de se termine pas non plus. Cet argument opérationnel suggère la définition dénotationnelle suivante du caractère strict :

DEFINITION

Une fonction f est dite stricte si et seulement si:

$$f \perp = \perp$$

Cette définition se généralise aisément aux fonctions à plusieurs arguments. Par exemple, si g est une fonction à trois arguments, alors g est stricte en son second argument si et seulement si :

$g\ a\ \perp\ c = \perp$

pour toutes les valeurs de a et de c.

Si une fonction n'est pas stricte, on dit qu'elle est **paresseuse**.

Il s'agit là d'un abus de langage, car l'évaluation paresseuse n'est qu'une technique d'implémentation d'une sémantique non stricte. Cependant, "paresseux" est si évocateur qu'il est souvent utilisé là où "non strict" aurait été correct.

Règles de conversion

Les règles de conversion données précédemment expriment des équivalences entre λ -expressions. Il est indispensable que ces équivalences possèdent leur contrepartie dans le monde dénotationnel. Par exemple, en utilisant la α-conversion, nous pouvons écrire :

$(\lambda x.+\ x\ 1) \leftrightarrow_{\alpha} (\lambda y.+y\ 1)$

Nous espérons que ces deux expressions possèdent la même signification, ou, plus précisément, dénotent la même fonction, de sorte que :

$Eval[\lambda x.+\ x\ 1] = Eval[\lambda y.+y\ 1]$

De manière générale, nous espérons que la conversion préserve le sens, ce que nous pouvons exprimer par :

$E_1 \leftrightarrow E_2$

implique :

$Eval[E_1] = Eval[E_2]$

En d'autres termes, si E_1 est convertible en E_2 alors le sens de E_1 est le même que celui de E_2. C'est une tâche difficile de prouver que l'énoncé précédent est toujours vrai, étant donné les règles de conversion et la fonction sémantique Eval.

Nous nous contenterons d'observer que la preuve est nécessaire, laissant celle-ci a [STOY 81].

Puisque les règles de réductions (β-réduction et η-réduction) forment un sous ensemble des règles de conversion, nous savons que :

$E_1 \rightarrow E_2$

implique :

$E_1 \leftrightarrow E_2$

et donc que :

$E_1 \rightarrow E_2$

implique :

$Eval[E_1] = Eval[E_2]$

Egalité et convertibilité

Dans la section précédente, nous avons vu que la conversion préserve l'égalité. Mais l'inverse est-il vrai? En particulier, l'égalité de deux expressions implique-t-elle leur inter-convertibilité? La réponse est négative, comme le montre l'exemple suivant. Considérons deux λ-abstractions que nous appellerons F_1 et F_2.

F_1 = (λx.+ x x)

F_2 = (λx.* x 2)

Clairement, F_1 ne peut être convertie en F_2 en utilisant les règles de conversion du λ-calcul. Pour un mathématicien, cependant, une fonction est une "*boîte noire*" et deux fonctions sont identiques si et seulement si elles donnent le même résultat pour chaque argument possible. Cette sorte d'égalité de fonction est appelée égalité extensionnelle. La fonction représentée par F_1 et celle représentée par F_2 sont (extentionnellement) égales, et donc, nous pouvons écrire :

$Eval[F_1] = Eval[F_2]$

Et pourtant F_1 et F_2 ne sont pas interconvertibles, et représentent néanmoins la même fonction.

En résumé :

si $E_1 \leftrightarrow E_2$

alors $Eval[F_1] = Eval[F_2]$

mais l'inverse n'est pas nécessairement vrai.

Nous pouvons donc voir la conversion comme une forme faible de raisonnement à propos de l'égalité d'expressions. Elle ne peut jamais nous laisser croire que deux expressions sont égales quand elles ne le sont pas, mais peut ne pas permettre de prouver l'égalité de deux expressions qui sont égales. De ce point de vue, la réduction est une forme d'inférence encore plus faible que l'égalité.

CHAPITRE 2: LANGAGES FONCTIONNELS

Maintenant que nous possédons les bases du λ-calcul, nous allons faire le tour de certains langages fonctionnels. Nous allons constater avec bonheur que tous reprennent les concepts du λ-calcul. Ils se différencient bien évidemment par la syntaxe, mais surtout par certaines caractéristiques que nous mettrons en évidence.
Nous avons classé les langages par leur ordre d'apparition. Cependant, nous conseillons au lecteur de commencer par le langage Scheme que nous avons plus développé.

Massachusetts Institut of Technologic

En fin nous ferons une étude plus poussée du langage Scheme du MIT, révision 4.
Nous avons obtenu une implémentation gratuite de ce langage sur le réseau Ethernet. Le logiciel est écrit par **Aubrey Jaffer** [5] et il est disponible à l'adresse électronique suivante :

```
altdorf.ai.mit.edu:archive/scm/scm4b4.tar.Z
```

Lisp

Lisp a été inventé à la fin des années cinquante comme un formalisme pour raisonner sur l'utilisation de certains types d'expressions logiques, appelées équations récursives, qui forment un modèle de calcul. Ce langage, conçu par J.McCarthy, est basé sur son article "*Recursive Functions of Symbolic Expressions and Their Computation by Machine*", paru en 1960.
Malgré son origine mathématique, Lisp est un langage de programmation pratique.
Un interprète Lisp est une machine qui exécute les processus décrits en langage Lisp. Le premier interprète Lisp fut réalisé par McCarthy avec l'aide de collègues et d'étudiants du Groupe Intelligence Artificielle du Laboratoire de Recherche en électronique du MIT et du Centre Informatique du MIT.

Le Lisp 1 Programmer's Manual date de 1960, et le Lisp 1.5 Programmer's Manual fut publié en 1962.

Lisp, dont le nom est un acronyme pour LISt Processing, a été conçu pour offrir des fonctionnalités de manipulation de symboles permettant de traiter des problèmes tels que la différenciation et l'intégration symbolique d'expressions algébriques. Il propose pour cela des objets nouveaux appelés atomes et listes, qui le distinguent radicalement de tous les autres langages de cette période.
Lisp n'a pas été le résultat d'un travail concerté. Au contraire, il a évolué de manière informelle à partir de l'expérience, répondant aux besoins des utilisateurs et aux contraintes pratiques de réalisation. L'évolution informelle de Lisp s'est poursuivie au cours des années et la communauté des utilisateurs de Lisp a traditionnellement été réticente à toute promulgation d'une définition "officielle" du langage. Cette évolution, alliée à la flexibilité et à l'élégance de la conception initiale, ont permis à Lisp, qui est derrière le Fortran, le plus vieux langage actuellement utilisé, de

s'adapter continuellement pour prendre en compte les idées les plus modernes en matière de programmation. Aussi Lisp est-il maintenant une famille de dialectes qui, tout en partageant la plupart des traits originaux, peuvent différer sensiblement entre eux.

Du fait de son caractère expérimental et de l'importance donnée à la manipulation de symboles, Lisp était à l'origine très peu efficace pour les calculs numériques, au moins par rapport à Fortran. Mais au fil des années, des compilateurs Lisp ont été développés qui traduisent les programmes dans un code machine très efficace.

Malgré cela, Lisp ne s'est pas encore débarrassé de sa vieille réputation de langage lent.

Nous invitons le lecteur désirant maintenant en savoir plus sur Lisp à se rendre au chapitre consacré à la présentation du langage Scheme, voisin moderne de Lisp. Scheme se différencie de Lisp par les points suivants :

- Liaison statique des noms de symboles,
- les fonctions peuvent prendre des fonctions en arguments,
- les fonctions peuvent rendre des fonctions,
- le langage s'est débarrassé de la plupart des constructions impératives de Lisp, comme PROG, REPLACA, REPLACD, etc.

Iswim

Iswim est l'acronyme de "If you See What I Mean"

Iswim fut le langage fonctionnel suivant, publié en 1966 par Landin dans son fameux article "*The Next 700 Programming Langages*".

Landin pensait qu'un langage fonctionnel ne devait pas nécessairement avoir une notation complexe. Le problème avec Lisp est l'usage intensif des parenthèses, qui rendent vite les programmes illisibles, les expressions lourdes et très imbriquées. Landin inventa pour palier à cet inconvénient la clause *Where*. Celle-ci est, dans la forme originelle de Iswim, récursive, autorisant les définitions croisées. L'ordre des définitions dans une clause *Where* n'a pas d'importance.

La syntaxe officielle de Iswim est une syntaxe abstraite composée d'un ensemble d'arbres d'analyse, plutôt qu'un ensemble de chaînes de caractères. Ce point autorise toutes les variantes dans les implémentations, laissant les concepteurs libres de la syntaxe.

Les fonctions dans Iswim sont des objets comme les autres, autorisant ainsi les fonctions à prendre des fonctions comme arguments, et à rendre de fonctions.

⊥ est appelé bottom. Il est présent dans la plupart des langages fonctionnels avec la même fonction.

Un élément essentiel du langage est le symbole ⊥ qui est utilisé comme le résultat de toutes les opérations invalides. Ce symbole peut être considéré comme la base de la pyramide des types de données. Toute fonction dont la valeur de l'un de ses arguments est ⊥ rend ⊥.

*Par exemple la définition : f(3) where f(n)=n*f(n-1)end fonctionne parfaitement mais entre d'après la sémantique opérationnelle, dans un processus infini. Avec la sémantique formelle, cette définition "est évaluée" en ⊥.*

Iswim utilise une sémantique formelle. Cela implique que certaines expressions ne sont pas évaluables et retournent ⊥. Par exemple, cet objet est "la sortie" d'un processus infini qui ne retourne jamais rien. C'est la valeur statique et dénotationnelle qui correspond à l'activité opérationnelle et dynamique ayant pour résultat "la non-terminaison", ou la non-valeur.

Iswim utilise (à l'instar de Scheme) la liaison statique des symboles. Les variables globales utilisées lors de la définition des fonctions constituent un environnement lié à la fonction elle-même. En d'autres termes, les variables globales prennent la valeur qu'elles ont au moment de la définition de la fonction, et non au moment de son évaluation.

Iswim utilise l'évaluation paresseuse. Les arguments d'une fonction sont évalués au moment de leur utilisation dans le corps de la fonction, et non avant l'appel de la fonction. La technique d'évaluation ne calcule que ce qui est strictement nécessaire.

Iswim possède des primitives de manipulation de liste, à rapprocher de celles de Lisp.
Du fait de l'évaluation paresseuse, Iswim autorise la construction de listes (et donc de structure) infinies. La tête de liste est fournie à la demande, sans évaluer la queue de liste. Cette approche est essentielle si on désire se rapprocher d'un modèle Data Flow.

Lucid

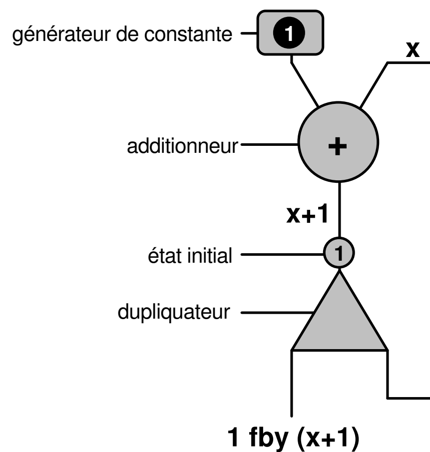
Lucid est un langage Data Flow. Il est un sur-ensemble de Iswim de Landin [5], est en reprend ainsi les principales caractéristiques. Il fut conçu en 1974 à l'Université de Waterloo au Canada. Ses auteurs sont WW.Wage et EA.Ashcroft [22]. Un évaluateur pLucid est écrit par A.Faustini [3] est disponible.
Les auteurs de Lucid sont partis de la constatation qu'il est difficile d'écrire des programmes basés sur la répétition en Iswim. Par exemple, construire un programme qui calcule la moyenne arithmétique de d'un flot d'entrée est impossible.
Pour palier à cette impossibilité, les auteurs de Lucid ont inventé l'opérateur `followed-by`. Par exemple, si on écrit :

fby est l'abréviation de followed-by.

```
x = 1 fby (x+1),
```

nous construisons la liste infinie des entiers naturels.
C'est par cet opérateur que Lucid est un langage Data Flow. En effet, le followed-by peut s'apparenter à un arc initialisé (le lecteur se référera au chapitre consacré aux graphes Data Flow) que nous pouvons représenter par :

Figure 5 : représentation avec un Data Flow de x = 1 fby (x+1)



Scheme

Cette section présente d'une façon plus approfondie le langage Scheme. Ce langage est défini par le MIT et en est à sa version 4. Scheme est un descendant du Lisp originel. Cependant, il y a un certain nombre de différences majeures, comme la portée statique des liaisons de variables, et le fait de considérer les fonctions comme toute autre donnée (cela autorise une fonction à retourner une autre fonction). Dans sa structure sémantique, Scheme s'apparente aussi bien à Algol 60, qu'au Pascal. Alors que Pascal permet de construire une hiérarchie de structure de données et d'imbrication de fonction. Scheme permet de décrire des "organismes" et des comportements dynamiques. Le programmeur Scheme jouit d'un pouvoir de description sans aucune mesure à ce qu'offre Pascal.

La liste, la structure élémentaire de Scheme y est pour beaucoup. En effet il n'y a pas de structure qui ne puisse être décrite à l'aide d'une liste. Par rapport au Pascal, l'atome des structures est toujours la même : la cellule. Là ou il faut invoquer trois fois le constructeur de liste, il faut en pascal définir une structure, la créer, la gérer et enfin la détruire. Il vaut mieux en Génie logiciel créer dix structures manipulées par cent fonctions, que 100 manipulées par 10. Il résulte de cela que les structures Pascal sont des pyramides qui doivent rester intactes au fil des mises à jour au lieu de s'y adapter.
Les concepts utilisés dans Scheme sont extrêmement simples et faciles à comprendre. Cependant, nous montrerons que parfois les notions sémantiques sous-jacentes seront extrêmement fines. Scheme ne dispose que de peu de moyens de construire des expressions composées, et de pratiquement aucune structure syntaxique. Un point encore : Scheme est basé sur le λ-calcul dans sa sémantique. Aussi nous y ferons référence chaque fois qu'il sera opportun de le faire.
Nous conseillons au lecteur qui désire se familiariser avec Scheme deux ouvrages. Le premier, de ABELSON & SUSSMAN, [13] est un cours sur la programmation utilisant comme langage Scheme. Le second est la norme publiée par le MIT [2].
Nous devons tout d'abord définir un certain nombre de convention syntaxique au niveau de ce rapport. Nous présentons cette section comme si nous disposions d'un interpréteur Scheme. Les commandes tapées par l'utilisateur seront précédées d'un ⇨ , alors que les valeurs retournées par l'interpréteur seront alignées sur la marge gauche.

Eléments de base

Voici des expressions Scheme :

```
⇨21
21
⇨(+ 1 2 3)
6
```

Type des données

Par opposition au **type de données manifeste** (type de la donnée déclaré), Scheme possède un **type latent**. Les types sont associés aux valeurs et non aux variables. Une variable pourra "au cours de sa vie" changer plusieurs fois de types automatiquement.

Appel de fonction

La syntaxe pour appeler une function est la même que dans tous les autres dialectes de Lisp : `(opérateur argument ...)`. L'opérateur prend la première position, et les arguments, les positions suivantes.
Cette syntaxe est équivalente à la syntaxe en λ-calcul, `(((opérateur) argument) ...)`, qui est sous une forme curryfiée.

Arguments

Les arguments des fonctions sont passés **par valeur**, ce qui signifie qu'ils sont évalués avant l'appel effectif de la fonction, que la fonction ait besoin ou non de la valeur de l'argument. Beaucoup de langages comme ML, C et APL passent leurs arguments par valeur.

Définitions

Scheme permet de nommer des objets :

```
⇨(define mon-objet 21)
#unspecified
```

Les fonctions qui ne retournent "rien" rendent un indicateur nommé #unspecified. Ainsi toute expression a une valeur.

donne à `mon-objet` la valeur `entier 21`. On peut donc maintenant utiliser `mon-objet` dans une expression, comme :

```
⇨(+ mon-objet 4)
25
```

Nous remarquons que les noms de variables peuvent utiliser le signe moins. En fait, tous les caractères sont utilisables pour définir des noms de variable. `Define` est le plus simple moyen d'abstraction du langage, car il nous permet de représenter, par des noms simples, des résultats d'opérations composées. Il serait extrêmement lourd de devoir rappeler et répéter leurs détails à chaque utilisation. Cette construction par morceau permet de construire un programme avec un ensemble de briques de manière incrémentale par interactions successives.

Cette possibilité d'associer des valeurs à des symboles oblige l'interprète à entretenir un **dictionnaire des symboles** composés des couples symbole-valeur. Ce dictionnaire est appelé **environnement** (plus précisément, dans ce cas, **environnement global**).

Nous verrons par la suite qu'il existe des environnements fils de l'environnement global.

Formes spéciales

On peut avoir l'intuition que dans l'exemple précédent, les choses ne se passent pas comme d'habitude. En effet nous avons vu que Scheme passe ses arguments par valeur, ce qui implique qu'ils soient préalablement évalués. Si tel était le cas, la commande :

```
⇨(define nom-objet 21),
```

produirait forcément une erreur, l'objet nom-objet n'étant pas défini, puisque c'est à lui, précisément, que l'on veut donner une valeur. Il y donc dans Scheme au moins une fonction qui ne se comporte pas comme les autres, en l'occurrence, la fonction `define`. Nous verrons par la suite que Scheme en a d'autres. C'est fonctions sont appelées **formes spéciales** et font partie du noyau du langage. Mais reprenons le cours de notre exemple.

Durée de vie des objets

L'objet `mon-objet` est crée et il occupe un emplacement quelque part dans la mémoire de l'ordinateur. Mais quand l'espace occupé sera-t-il récupéré ? La réponse est jamais. Par contre, la valeur de l'objet, ici l'entier 21, pourra être détruit si par exemple on tape la ligne :

```
⇨(define nom-objet "une chaîne")
#unspecified
```

Ce mécanisme est décrit plus précisément dans le chapitre consacré à l'implémentation d'un interpréteur Scheme.

Maintenant, l'entier 21 n'est plus attaché à `mon-objet` est peut être récupéré. La libération de l'espace occupé par l'entier 21 sera faite automatiquement lorsqu'il n'y aura plus assez de mémoire disponible par un mécanisme dit de "ramasse miette" (*garbage-collector*).

Evaluation des expressions

L'interprète Scheme utilise lui-même une fonction pour évaluer une forme :

1. évaluer les sous-expressions de la forme,
2. appliquer la fonction, valeur de la sous-expression de gauche (opérateur), aux arguments, valeurs des autres sous-expressions (les opérandes).

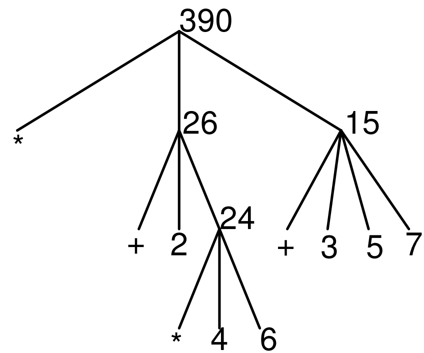
Nous voyons qu'avant d'appeler effectivement l'opérateur, il faut évaluer chacun des éléments de la forme. La règle d'évaluation est donc nécessairement récursive. L'évaluation de l'opérateur lui-même permet de placer à la place de l'opérateur une expression composée qui rend une fonction.

Par exemple :

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

impose d'appliquer la règle d'évaluation à quatre formes différentes. La figures suivantes donne une image de ce calcul sous forme d'un arbre. Chaque forme est représentée par un noeud dont les branches correspondent à l'opérateur et aux opérandes.

Figure 6: évaluation par accumulation d'arbre.



Cet arbre se parcourt en suivant le sous-arbre de gauche, puis le sous-arbre de droite, et enfin, la racine. Les résultats intermédiaires sont conservés dans chaque noeud.

Quotation

La quotation est un mécanisme indiquant à l'interpréteur de ne pas évaluer l'expression qui suit. Par exemple :

```
⇨ '(+ 1 2)
(+ 1 2)
```

retourne la liste formée de l'identificateur +, suivi de l'identificateur 1 et de l'identificateur 2.

```
⇨ '()
#NULLOBJECT
```

Listes

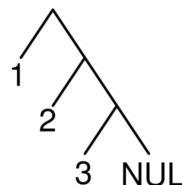
La **liste** est la structure essentielle de Scheme. Avec la liste, tout le langage est construit. Une liste se compose d'une **tête** de liste, et d'une **queue** de liste. La tête et la queue de liste peuvent être eux-mêmes des listes. Il existe trois opérateurs pour manipuler les listes, `CONS`, qui agit comme un constructeur, `CAR` qui retourne la tête de la liste, et `CDR` qui retourne sa queue.

Nous avons vu dans le chapitre consacré au λ-calcul des opérateurs similaires nommés `CONS`, `HEAD` et `TAIL`. Nous avons vu que ces opérateurs pouvaient être définis en λ-expressions. Il en est de même avec Scheme et `CONS`, `CAR` et `CDR` qui peuvent être écrit en Scheme. Nous verrons cela après avoir parlé des fonctions en Scheme. Mais revenons à nos listes. Nous voulons former la liste des trois premiers entiers. Nous écrivons :

```
⇨(cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

Une représentation schématique peut être donnée :

Figure 7 : représentation schématique de la liste (1 2 3)

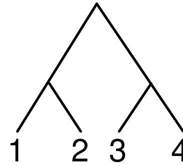


Il est important de noter la présence d'un élément spécial : `NUL`. Il représente une liste vide, et pour conserver une structure homogène à la liste, il est le dernier élément.
A l'aide des listes, on peut aussi construire la structure suivante:

```
⇨(cons (cons 1 2) (cons 3 4))
((1.2).(3.4))
```

qui peut être représenté schématiquement par :

Figure 8: structure "complexe"



Il est donc ainsi possible à l'aide des simples listes de former des structures complexes telles que les couples, les arbres binaires ou non, etc.
Une autre façon de fabriquer des listes et des structures et la **quotation**. Elle indique à l'interpréteur de ne pas évaluer ce qui va suivre. Par exemple :

```
'(1 2 3)
```

est la liste formée de 1 puis 2 et 3.

Représentation externe

Nous avons vu au cours des exemples précédents que l'interpréteur donne toujours en retour une expression. Cette expression affichée est la **représentation externe** d'un objet. Une liste est représentée par un ensemble de valeurs entre parenthèses. Un couple est représenté par deux expressions entre parenthèses séparées par un point. Si nous tapons:

```
⇨+
#procedure
```

Dans ce cas, la valeur de retour dépend de l'interpréteur Scheme utilisé.

l'intepréteur nous retourne `#procedure` qui indique que l'objet évalué est une fonction.

Définition de fonction

Pour définir une fonction, il faut utiliser l'opérateur spécial `lambda` (à rapprocher du λ du λ-calcul). Cet opérateur attend au moins deux arguments :

- une liste d'arguments formels,
- une instruction (ou plusieurs)

Par exemple :

```
⇨(lambda (a b) (+ a b 10))
#procedure
```

défini une "*fonction à deux arguments qui retourne la somme des deux arguments et de dix*". Bien évidemment, il est possible d'appliquer une telle fonction à des valeurs, comme :

```
⇨((lambda (a b) (+ a b 10)) 3 4)
17
```

Nous remarquons que l'opérateur `lambda` de Scheme agit de la même manière que l'opérateur λ du λ-calcul.
Heureusement que nous avons l'opérateur `define` qui va nous permettre de nommer une fonction et de la conserver dans l'environnement global :.

Par exemple, la fonction d'incrémentation peut s'écrire :

```
⇨(define inc (lambda (num) (+ num 1)))
#unspecified
```

Nous pouvons maintenant utiliser la nouvelle fonction `inc` :

```
⇨(inc 100)
101
```

Il existe une syntaxe simplifiée utilisant l'opérateur `define`.

```
(define (f a b) (+ a b))
```

définit la fonction f à deux arguments, a et b, qui retourne la somme de ses arguments.

Fonctions : des objets comme les autres

Scheme a une particularité très intéressante qui n'existe pas dans les autres définitions de Lisp : les fonctions sont considérées comme tous les autres objets.

Elles peuvent être passées en paramètres, ou être retournées par une fonction.

Considérons par exemple la dérivée. La proposition "La dérivée de x^3 est $3x^2$" signifie que la dérivée de la fonction dont la valeur en x est x^3 est une autre fonction, plus précisément la fonction dont la valeur en x est $3x^2$.

Plus généralement, la "dérivation" peut être vue comme un opérateur qui appliqué à une fonction `f`, donne pour résultat une autre fonction `Df`. Pour décrire la dérivée, nous pouvons dire que si `f` est une fonction et `dx` un certain nombre, alors la dérivée `Df` de `f` est la fonction dont la valeur au point `x` est donné, lorsque `dx` tend vers zéro par :

$$Df(x) = \frac{f(x+dx)-f(x)}{dx}$$

En utilisant `lambda`, cette formule déclarative donne la fonction

```
(lambda (x)
  (/ (- (f (+ x dx))(f x))
     dx)))
```

où `dx` est un nombre. Nous pouvons utiliser ceci pour exprimer l'idée de la dérivée elle-même dans la fonction

```
⇨(define dérivée (lambda f dx)
⇨ (lambda (x)
⇨   ((/ (- (f (+ x dx))(f x))
⇨       dx))))
```

Cette fonction prend pour argument une fonction `f` et retourne une fonction (produite par `lambda`) qui appliquée à un nombre `x`, fournira une valeur approchée de la dérivée de la fonction `f` en `x`.

Nous pouvons par exemple utiliser notre fonction `dérivée` pour calculer une valeur approchée de la dérivée de la fonction `cube` en `5` (la valeur exacte est `75`) :

```
⇨(define cube (lambda x)(* x x x))
⇨((dérivée cube .001) 5)
75.015
```

L'opérateur de forme ci-dessus est lui-même une forme, car la fonction à appliquer à `5` est la valeur de `dérivée` appliquée à `cube`.

CONS, CAR et CDR en Scheme

Nous supposons que les opérateur CONS, CAR et CDR ne sont pas définis dans cet interpréteur. x et y sont respectivement la tête et le queue de la liste, z est la liste elle-même.

Nous avons dit précédemment que ces opérateurs ne sont pas essentiels. Cela signifie qu'ils peuvent être **redéfinis en Scheme**, avec des fonctions. Voici ces fonctions :

```
⇨(define (cons x y)
⇨  (define (aiguiller m)
⇨    (cond((= m 0) x)
⇨          ((= m 1) y)
⇨          (else (display "error"))))
⇨  aiguiller)
#unspecified
⇨(define (car z)(z 0))
⇨(define (cdr z)(z 1))
#unspecified
```

La valeur retournée par `(cons x y)` et une fonction, à savoir la fonction interne `aiguiller` qui a un argument et retourne `x` ou `y` selon que l'argument vaut `0` ou `1`. `(car z)` est défini comme l'application de `z` à `0`. Et, désormais, si `z` est la fonction construite par `(cons x y)`, alors `z` appliquée à `0` produira `x`. Il en est de même pour `cdr`.
Cette construction est valide quoique totalement inefficace. Mais, hormis la beauté du geste, elle montre que les fonctions `cons`, `car` et `cdr` ne sont pas essentielles, comme l'est par exemple la fonction `define`.

Variables libres, liées et environnements.

Nous avons déjà abordé le sujet au cours de notre introduction au λ-calcul. Rappelons simplement que dans une fonction, il existe deux sortes de variables : les **variables liées** et les **variables libres**.

- les variables liées sont les arguments de la fonction,
- les variables libres ne font pas partie des arguments. Il faut trouver leur valeur dans l'environnement global.

Lorsque nous écrivons :

```
((lambda (a b) (+ a b c)) 3 4)
```

les variables `a` et `b` sont liées car elles font partie de la liste des arguments. Par contre la variable `c` doit être trouvée ailleurs. Dans ce cas, l'interpréteur la cherchera dans l'environnement global. Si `c` a été définie par un `(define c 31)`, par exemple, le résultat sera `38`. Si `c` n'a pas été précédemment définie, l'interpréteur affichera un message d'erreur indiquant qu'il ne trouve pas la variable `c`.
Mais que se passe-t-il lorsque nous entrons l'expression :

```
⇨((lambda (a b)              ①
⇨ ((lambda (a b) (+ a b))    ②
⇨   (+ a b)
⇨   (* a b))) 3 4)
```

Il y a visiblement un conflit entre les variables `a` et `b` liées de la fonction `lambda` et les variables liées `a` et `b` de la seconde.
En fait, l'opérateur `lambda` crée un **environnement local** dans lequel il place les arguments formels ainsi que toutes les variables crées dans le corps de l'expression. Cet environnement a pour parent l'environnement global ou un autre environnement crée lui aussi par une expression `lambda`. Dans l'exemple, l'expression ① crée un environnement dont le père est l'environnement global, alors que l'expression ② crée un environnement dont le père est l'environnement crée par ①. Il y a ainsi une

hiérarchie des environnements. Chercher un symbole dans un environnement revient à le chercher dans l'environnement courant, puis dans tous les environnements parents.
Lors de l'évaluation d'une expression `lambda`, un environnement est crée. Les arguments formels y sont ajoutés avec leur valeur au moment de l'évaluation. Le corps de l'expression `lambda` y cherchera désormais tous les symboles utilisés. Donc dans le cas de ① et ②, il n'y aura pas de conflit, et la valeur retournée par l'interpréteur sera `19`.

Fonctions récursives

Considérons la fonction factorielle, définie par : :

`n! = n.(n-1).(n-2)...3.2.1`

Il y a plusieurs façons de calculer une factorielle. L'une d'entre elles part de l'observation que `n!` est égal à n fois `(n-1)!` pour tout entier positif `n` :

`n! = n.((n-1).(n-2)...3.2.1) = n.(n-1)!`

Nous pouvons donc calculer `n!` en calculant `(n-1)!` et en multipliant le résultat par `n`. Sachant que `1!` est égal à `1`, ceci se traduit directement par la fonction :

```
⇨(define fact (lambda (n)
⇨ (if (= n 1)
⇨    1
⇨    (* n (fact (- n 1))))))
```

Montrons maintenant le déroulement de l'exécution de `(fac 5)`:

```
(fac 5)
(* 5 (fac 4))
(* 5 (* 4 (fac 3)))
(* 5 (* 4 (* 3 (fac 2))))
(* 5 (* 4 (* 3 (* 2 (fac 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
```

Calculons maintenant la factorielle d'une autre façon. Une règle de calcul de `n!` peut consister à multiplier d'abord `1` par `2`, ensuite multiplier ce résultat par `3`, puis par `4`, et ainsi de suite jusqu'à `n`.
Plus concrètement, nous mettons à jour le produit courant et un compteur allant de `1` à `n`. Nous pouvons décrire le calcul en disant que le compteur et le produit varient en même temps d'une étape à la suivant en suivant la règle :

```
produit    ← compteur x produit
compteur   ← compteur + 1
```

en précisant que `n!` est la valeur du produit lorsque le compteur dépasse n.

Cette description donne aussi une fonction de calcul d'une factorielle :

```
⇨(define fac (lambda (n)
⇨ (fac-iter 1 1 n))
⇨(define (fac-iter produit compteur max-compteur)
⇨ (if (> compteur max-compteur)
⇨   produit
⇨   (fac-iter (* compteur produit)
⇨             (+ compteur 1)
⇨             max-compteur)))
```

Comme précédemment, visualisons le processus d'évaluation dans le calcul de `(fac 5)` :

```
(fac 5)
(fac-iter   1 1 5)
(fac-iter   1 2 5)
(fac-iter   2 3 5)
(fac-iter   6 4 5)
(fac-iter  24 5 5)
(fac-iter 120 6 5)
```

120

Comparons les deux processus.
Ils calculent la même fonction mathématique en un même nombre d'opérations. Il faut aux deux processus un même nombre de pas de calcul pour arriver au résultat. Mais le premier programme utilise manifestement une zone de stockage des résultats intermédiaire différent du premier.
C'est un processus récursif. Cette forme de programmation existe dans tous les langages évolués. Ceux-ci utilisent une pile pour transmettre les arguments à une fonction. Dans la plupart des microprocesseurs, cette pile est gérée automatiquement d'une manière extrêmement efficace directement par des instructions machines `PUSH` et `POP`, et une zone mémoire est allouée quelque part dans le mémoire de l'ordinateur à cet effet.

Les DSP qui sont des processeurs de calcul n'ont pas de gestion automatique de pile. Les fonctions récursives ne pourront pas être implémentées sur de tels processeurs.

L'appel à une fonction provoque l'empilement des paramètres et de l'adresse de retour. Lors du retour de la fonction, le processeur restaure la pile dans l'état où elle était avant l'appel de la fonction.
Ce second processus est lui itératif. Il s'exécute dans un espace constant, transmettant tous les paramètres à chaque fois.

Récursion de queue

Ce terme est une traduction de l'anglais "*tail-recursion*". Considérons la fonction suivante :

```
⇨(define map (lambda (function list)
⇨ (if (not (eq? list ()))
⇨   (begin
⇨     (function (car list))
⇨     (map (cdr list))))))
```

Cette fonction applique une fonction à tous les éléments d'une liste. Comme on peut le voir, elle est définie de manière récursive puisque `map` appelle `(map (cdr list))`.
Mais considérons la traduction de cette fonction dans un pseudo langage ne possédant pas la possibilité des fonctions récursives :

```
PROCUDURE MAP(function, list)
BEGIN
  LABEL begin_proc:
    IF (LIST ≠ ()) THEN
      function CAR(list)
      list = CDR(list)
      GOTO begin_proc
    ENDIF
ENDPROC
```

La récursion terminale n'utilisant pas de pile peut être implémentée sur des processeurs de type D.S.P.

Cette fonction est parfaitement décrite sans utiliser la récursion. Elle s'exécutera comme une itération, dans un espace mémoire constant. Ceci est possible car la fonction map utilise la récursion de queue. En effet la dernière fonction appelée dans map est map elle-même. La valeur retournée par map_1 est le résultat retourné par map_{n+1}. L'interpréteur peut détecter cette forme de récursion et la transcrire en itération.

Action sur les environnements

Scheme offre un certain nombre de fonctions permettant de manipuler les environnements. Ces fonctions peuvent être remplacées par des λ-expressions à part deux : `set!` et `define`.

Define

Cette fonction permet de définir des symboles dans l'environnement en cours. Si le symbole existe, sa valeur est remplacée, sinon, il est créé. L'utilisation la plus naturelle de `define` est dans le *top-level* (ailleurs que dans le corps d'une λ-expression).
Cette fonction ne fait pas partie du λ-calcul. Elle agit par définition par effet de bord et en séquence. Dans l'environnent global, c'est justement l'effet recherché. Mais il y une utilisation plus subtile de define : dans le corps des λ-expressions. Là, elle permet aussi de créer des effets de bord. Par exemple :

```
(define une_fonction (lambda ()
  (define a 3)
  (display a)
  (define a 300)
  a))
```

Le `lambda` crée un nouvel environnement. Le premier `(define a)` crée `a` dans le nouvel environnement. Le second modifie la valeur liée à `a`. Cela signifie que `a` n'a pas toujours la même valeur dans le corps de `une_fonction`.
Notons qu'il y a une syntaxe étendue de define qui permet de créer des fonctions d'une manière plus lisible :

```
(define (nom_procedure arguments ...) corps_de_la_fonction)
```

qui permet d'écrire :

```
(define (function a b c)
  (display a)
  (+ b c))
```

⇔

```
(define function (lambda (a b c)
  (display a)
  (+ b c)))
```

Set! Set! permet de modifier la valeur d'un symbole dans l'environnement courant. Si le symbole n'existe pas, il y a une erreur. Cette fonction est généralement utilisée dans le corps des λ-expression pour modifier la valeur des arguments.

Let Cette fonction permet de définir des variables à l'intérieur d'une fonction. Là où nous écrivons en c :

```
int a_function (int x) {
  int temp;

  temp = another_function();
  return temp+x;
}
```

nous écrivons en Scheme :

```
(define a_function (lambda (x)
  (let ((temp (another_function)))
    (+ temp x))))
```

Le `let` permet de créer une variable temporaire. Cette variable "**ne peut être vue**" qu'a l'intérieur du corps de `a_function`. La syntaxe exacte de `let` est :

```
let  v1 = E1
     v2 = E2
     ...
     vn = En
in E
```

où les v_i sont des variables et E, $E_1, ..., E_n$ sont des expressions Scheme.
Le `let` peut être remplacé par une λ-expression :

Le fait que le let soit λ-définissable nous permet de lui appliquer les mêmes règles que les λ-expressions

$$(\text{let } ((v_1\ E_1)\ (v_2\ E_2)\ \ldots\ (v_n\ E_n))\ E)$$

$$\Leftrightarrow ((\text{lambda } (v_1\ v_2\ \ldots\ v_n)\ E)\ E_1\ E_2\ \ldots\ E_n)$$

Cette traduction en λ-expression apporte certaines précisions indispensables quant à la sémantique du `let`.
Rappelons que l'interpréteur évalue les paramètres d'une application avant d'évaluer l'application elle-même. Dans le cas présent, les E_i seront évalués en premier, puis

la λ-expression sera évaluée avec les résultats des E_i. Par exemple, un E_i ne peut pas utiliser l'un des v_i pour s'évaluer :

```
(let ( ( a 3)
       (b (+ a 10))
  (* a b))
```

provoque une erreur si a n'est pas défini dans l'environnement parrent car `a` n'est pas défini dans `(b (+ a 10))`.
En clair, cela signifie qu'aucun E_i ne doit faire référence à un v_i.

Letrec La syntaxe d'une expression letrec est similaire à celle du let :

```
let  v1 = E1
     v2 = E2
     ...
     vn = En
in E
```

`Letrec` est la contraction de *let recursivly*. Cet opérateur introduit des liaisons éventuellement récursives pour les variable v_i. La différence entre `let` et `letrec` est que la portée de v_i n'est pas limité à `E` mais s'étend à tous les E_i.
`Letrec` permet par exemple la définition suivante :

```
(letrec (  (f (lambda (x)(+ x 10))
           (g (lambda (y) (f y)))
  (g 15))
```

Bien que cette expression n'ait pas beaucoup de sens, elle montre la possibilité de définir les v_i de manière récursive.
Le letrec est λ-calculisable. son expression est alors :

```
(letrec (  (v1 E1)
           ...
 E)
```

⇔

```
(let ( (v1 #undefined)
       ...)
  (let ( (temp1 E1)
         ...)
    (set! v1 temp1)
    ...)
  E)
```

où `#undefined` représente la valeur "*symbole non défini*", et `set!` la fonction qui modifie la valeur d'une variable (cf.: plus bas).

Notons qu'une expression letrec peut être dangereuse. Par exemple l'expression :

```
(letrec ((x (+ (x y))            ①
         (y (+ (x y))) x)
```

On attend légitimement comme résultat 0, qui est la seule solution du système d'équation :

$$\begin{cases} x = x + y \\ y = x + y \end{cases}$$

Mais **Scheme ne sait pas résoudre les équations**, il permet de mettre en oeuvre un processus opératoire pour arriver à un résultat. Si on tentait d'évaluer l'expression ① , l'interpréteur entrerait dans une évaluation infinie. C'est pour cette raison que certains interpréteurs Scheme n'autorisent que les λ-expressions comme E_i. d'un `letrec`.
Pour ces raisons, notre interpréteur aura un commutateur laissant le choix à l'utilisateur quant à la manière de traiter les `letrec` : les interdire, les autoriser seulement avec des λ-expressions dans les E_i, ou ne faire aucun contrôle des E_i.

Let* Cet opérateur a une syntaxe identique a celle de `let` et de `letrec`. Cependant il permet à un E_i de faire référence aux v_i situés au-dessus de lui. Cette particularité permet de créer des effets de bords.
La λ-expression équivalente est :

```
(let* (  (v1 E1)
         (v2 E2)
         ...)
  E)
(let ((v1 E1))
  (let* (  (v2 E2)
           ...)
  E))
```

Autres fonctions Nous donnons maintenant un certain nombre de fonctions λ-définissables dites de confort, ou "sucre syntaxique". Une liste plus complète est donnée dans la définition du langage publiée par le MIT [2].

- **begin** : permet d'effectuer une séquence d'instruction dans un même bloc.

```
(begin E1 E2 ... En)
⇔
((lambda () E1 E2 ... En))
```

- **cond** : cette fonction permet de faire une sélection par cas. sa syntaxe est :

```
(cond (  (test1 clause1)
         (test2 clause2)
         ...
         (testn clausen)
         (else clause)))
```

⇔

```
(if (test₁) (begin clause₁)
  (if (test₂) (begin clause₂)
    ...
    (if (testₙ) (begin clauseₙ)
      clause)))
```

où `clause`$_i$ est une liste d'expressions.

Il y a d'autres fonctions λ-définissables comme and, or, case, do, etc. Elles sont toutes λ-définissables et donc conforme à la théorie du λ-calcul.

Evaluation de Scheme

Comme le lecteur à pu le constater, le langage Scheme s'appuie énormément sur le λ-calcul. Sa sémantique est très pure. Il possède peu de formes spéciales (define, set!, if, lambda), les opérateurs du langage étant pour la plupart λ-définissables.
Dans le cadre de notre projet, nous avons à réaliser un interprète de λ-expressions. Nous tournerons vers un interprète Scheme offrant la possibilité de "régler" son degré de fidélité au λ-calcul, en interdisant certaines formes spéciales.
Notre but est de programmer des processeurs du type D.S.P. Ces processeurs sont orientés dans le traitement du signal, et pour cette raison n'offre pas toutes les possibilités des processeurs a usage général : peu d'espace d'adressage, pas de pile pour la plupart, communications vers l'extérieur réduites au minimum, etc. C'est limitations se traduiront au niveau du langage de programmation. Le fait qu'il ait peu de mémoire vive interdira, par exemple, les opérateurs cons, car et cdr. Le fait qu'il n'y ait pas de pile interdira la définition de fonctions récursives (si la récursion n'est pas du type récursion de queue ou *tail-recursion*).
Mais dans l'état actuel de nos recherches, nous ne savons pas par exemple quel D.S.P. sera choisi, nous ne savons pas quelles limitations apporter au langage. Celui-ci devra par conséquent être hautement configurable permettant ainsi de choisir précisément les opérateurs autorisés.

PARTIE 2: LES GRAPHES DATA FLOW

Dans cette partie, nous présentons une étude des graphes Data Flow. Nous n'avons pu par manque de temps faire personnellement la totalité de cette étude ; nous nous appuyons ici sur les travaux de J.Demartini, et plus particulièrement sur ses implémentation, LUXIFERE [15] et plus récemment JEZABEL. La première implémentation fut faite en langage C. La seconde, qui met en évidence les différentes formes théoriques de Data Flow, est écrite en langage Scheme.
Nous conseillons aussi au lecteur le livre de JA.SHARP[16].

MODELES THEORIQUES

Avant d'étudier les différents modèles théoriques, nous nous intéressons en premier lieu à la théorie des ordinateurs. En effet, les graphes Data Flow se justifient par le fait que les modèles des ordinateurs se révèlent peu efficaces et inadaptés pour certaines applications. Certains auteurs prétendent même que le modèle des ordinateurs, basé sur le modèle de Von Neuman, est structurellement mal construit, favorisant les erreurs de programmation. Ce point est d'autant plus sensible que l'application développée est importante.
Les graphes Data flow s'inscrivent dans une évolution radicale de ces modèles d'ordinateur, et tendent vers des architectures parallèles. Ces modèles rompent définitivement avec le séquencement des instructions pour laisser émerger des modèles basés sur la répartition des tâches et des ressources.

Théorie des ordinateurs

Von Neumann

Actuellement, il existe des machines construites autour d'architectures nouvelles, comme les réseaux HyperCube.

Il y a encore quelques années, tous les ordinateurs étaient basés sur le modèles théorique de Von Neumann. Ce modèle repose sur la présence d'une mémoire et d'un processeur. Le programme (suite d'instruction) est présent dans cette mémoire, ainsi que les données (ici nous faisons volontairement abstraction des autres dispositifs, comme les entrées/sorties, qui compliquent la compréhension sans modifier les principes).
Le processeur lit ses instructions dans la mémoire en transférant dans ses registres internes l'instruction à exécuter. Cette instruction est "pointée" par le compteur ordinal. Après lecture de l'instruction, ce compteur est automatiquement incrémenté pour "pointer" sur l'instruction suivante. Ensuite, il décode l'instruction et l'exécute.
Les données se déplacent de la mémoire où elles sont stockées vers les registres internes où elles sont manipulées, puis des registres vers la mémoire pour y être rangé.
La plus grande partie du temps est pris par des transferts d'information d'un composant de la machine vers un autre.
Cette architecture se reflète dans les langages de programmation traditionnels par l'existence de l'affectation, et des variables. L'affectation représente le transfert des registres du processeur vers la mémoire. Toutes les références à des variables représentent quant à elles le flux inverse, de la mémoire vers le processeur.
Le compteur ordinal est lui aussi représenté dans les langages de programmation conventionnels par des instructions de contrôle comme le *goto*, les instructions conditionnelles et les boucles. Au lieu que le compteur ordinal soit incrémenté, sa valeur est directement ou indirectement modifiée. Ainsi, ces instructions modifient le cours du programme selon certaines conditions. C'est aussi une forme d'affectation, celle-ci opérant sur le compteur ordinal.

L'appel de fonction est radicalement différent. en effet il stock un point de retour. Le cours du programme n'est pas modifié.

Ce modèle est basé sur la notion d'état. L'état est représenté par la valeur du compteur ordinal ainsi que le contenu de la mémoire, et cela à chaque instruction. Le programmeur a l'entière responsabilité du déroulement du programme, donc de l'état de la machine.

Sur un Ibm Pc de 10 Mo avec un disque dur de 200 Mo.

De nos jours, le programmeur est couramment directement ou indirectement responsable de dix millions de cellules de mémoires, sans parler des 200 millions de cellules de la mémoire auxiliaire! Les machines actuelles tentent de réduire cette responsabilité en allouant des segments de mémoire à un programme. Cela est utile pour les autres processus qui ont moins de chance d'être interrompus par l'écrasement sauvage de leurs données, mais ne résout rien.

Le parallélisme dans de telles architectures peut être implémenté, mais à quel prix! Soit les systèmes d'exploitation deviennent de véritables usines, soit le programme est complètement illisible, parsemé d'instructions exotiques tentant de paralléliser plusieurs tâches ou co-routines entre elles.

Modèles Data Flow

Le modèle Data Flow tente d'apporter des solutions aux problèmes liés à l'architecture de Von Neumann :

- dans un programme Data Flow, l'ordre des opérations n'est pas explicitement décrit par le programmeur, mais induit par l'interdépendance des données,
- dans un programme Data Flow, le séquencement des opérations est effectué en fonction de l'interdépendance des données et de la disponibilité des ressources.

Le modèle Data Flow est intimement lié à une notation graphique des programmes où les noeuds représentent des opérations et les arcs les dépendances et les flots de données. En général, un programme peut être représenté directement par un graphe. Cependant, nous verrons que certaines constructions non aucun sens.

Graphe simple

Construisons notre premier graphe :

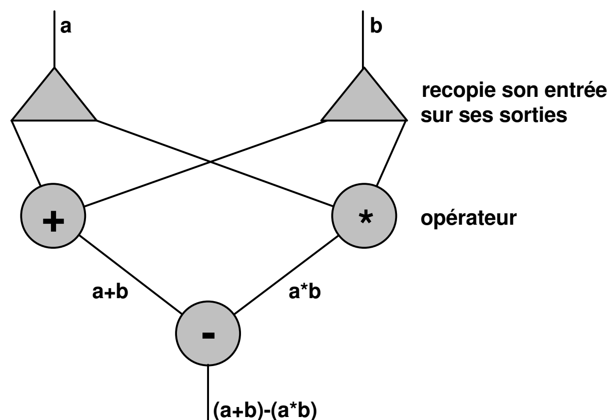


Figure 9 : programme pour calculer la différence entre la somme et le produit de deux nombres.

Nous avons deux manières de considérer ce graphe, par le haut ou par le bas. Par le haut, dès que des données sont présentent en a et b, elles circulent le long des arcs. On peut aussi considérer le graphe par le bas : dès que l'on réclame le résultat, le graphe propage la demande de donnée vers le haut. Ce mécanisme de propagation est souvent illustré avec des jetons qui circulent le long des arcs. Les jetons emportent avec eux une valeur. Cette valeur est souvent un nombre, mais elle peut en fait être n'importe quoi. Le déclenchement des opétateurs se fait par les jetons.

Respectivement piloté par les données et piloté par la demande.

Nous avons affaire aux deux principaux modes de fonctionnement des graphes Data Flow, le mode *data-driven* et le mode *demand-driven*.

En mode *data-driven*, le fonctionnement est celui-ci :

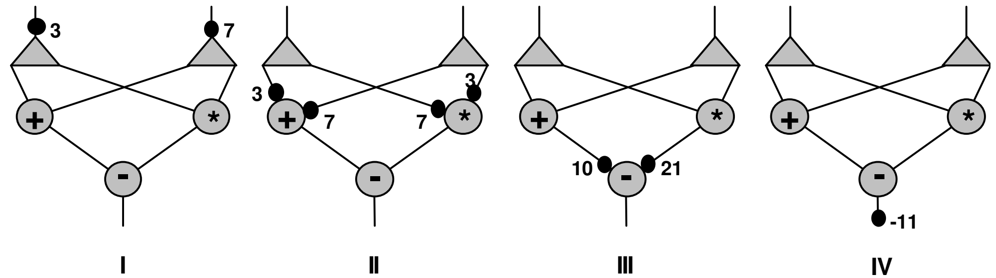


Figure 10 : processus opératoire en mode *data-driven*.

Les données se propagent le long des arcs. Lorsqu'un opérateur a des données sur toutes ses entrées, il peut "opérer", et fournir une donnée sur sa sortie.
En Mode *demand-driven* les choses sont similaires, la demande pouvant être représentée par un jeton se propageant de bas en haut.
Sur un graphe simple comme celui-ci, les choses se déroulent comme prévues.

Contre-réaction

Mais les schémas réels font souvent appel à des contre-réactions. Que ce passe-t-il lorsque l'une des entrées d'un opérateur dépend directement ou indirectement de sa sortie ? Par exemple :

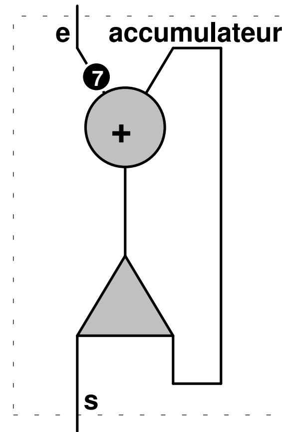


Figure 11 : accumulateur ; problème de la contre-réaction..

A l'instant initial, si nous plaçons un jeton sur `e`, l'opérateur d'addition ne pourra pas effectuer son calcul, un jeton lui manquant sur sa seconde entrée.
Certains auteurs interdisent purement et simplement cette configuration, n'autorisant que les schémas acycliques, comme JA.SHARP [16]. Cela résout clairement le problème, mais interdit du même coup toutes sortes de contre-réaction, ce qui paraît peu vraisemblable.

Valeur initiale Une solution consiste à placer autoritairement dans tous les arcs du graphe une valeur initiale. L'exemple devient :

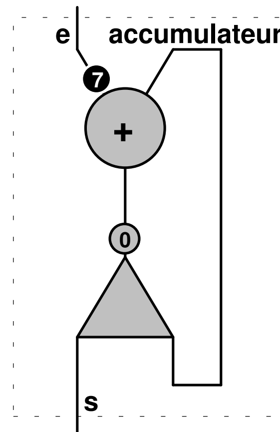


Figure 12 : accumulateur initialisé avec la valeur 0.

La présence de l'arc initialisé résout le problème de la famine de jeton. Le processus d'opératoire devient alors :

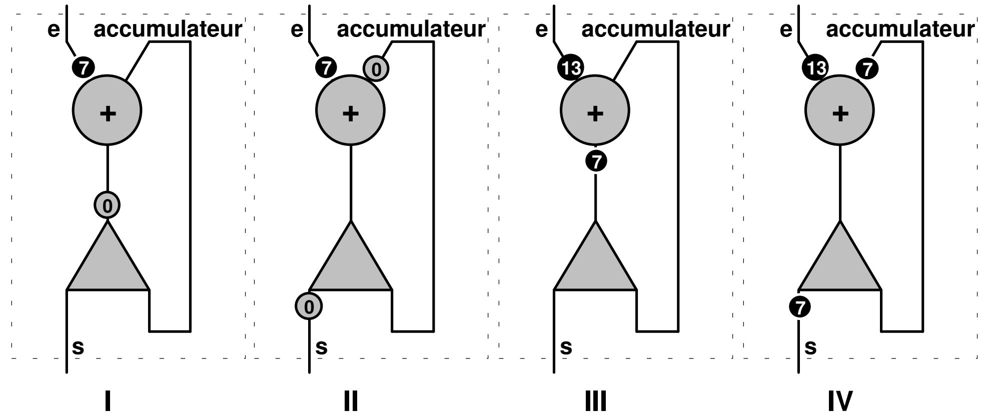


Figure 13 : accumulateur en fonctionnement.
I : présentation du jeton 7
II : circulation des jetons
III : consommation du jeton 0, présentation du jeton 13
IV : circulation des jetons.

Data Flow synchrone Un graphe Data Flow est dit synchrone lorsque chaque opérateur consomme et produit toujours le même nombre de jeton quelque soit la valeur des jetons en entrée.

Ce synchronisme est une donnée capitale. En effet, l'évaluation d'un graphe synchrone est statique ; le graphe peut être compilé, puis exécuté par la suite. On de plus peut "mesurer" le temps de calcul d'un tel graphe. Si le graphe n'est pas synchrone, il est impossible de calculer ce temps.

Pour donner un exemple, un opérateur asynchrone bien connu est le `if`. Imaginons un `if` en tant qu'opérateur Data Flow :

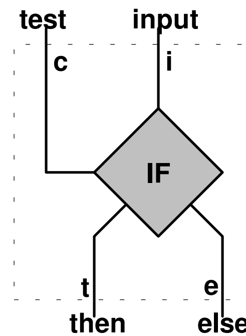
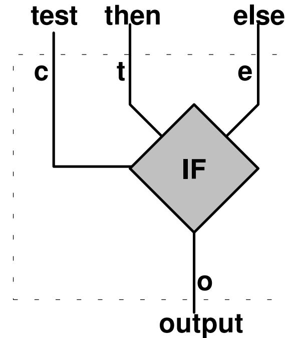


Figure 14 : opérateur `if` asynchrone.

Selon la valeur du jeton `c`, le jeton `i` va en `t` (valeur `TRUE` de `c`) ou en `e` (valeur FALSE de `c`). Ce comportement est asynchrone, puisqu'on ne peut pas déterminer à l'avance la valeur de c.
Pour rendre cet opérateur synchrone, il faut nécessairement que les clauses THEN et ELSE soient toujours évaluées. L'opérateur est alors :

Figure 15 : opérateur `if` synchrone.



Selon la valeur du jeton présent à l'entrée `c`, le jeton `t` (valeur `TRUE` de `c`) ou le jeton `e` (valeur `FALSE` de `c`) est reproduit en `o`. Ce fonctionnement est parfaitement synchrone.

Input Output Rate

Dans le modèle de graphe Data Flow développé par Lee et Messerschmitt de l'Université de Berkeley, les opérateurs peuvent produire un nombre de jeton différent du nombre de jeton consommé. Pour donner un nom a ce concept, nous le battions IOR ou *Input Output Rate*, qui représente le rapport du nombre de jetons consommés au nombre de jetons produits.
Le graphe Data flow reste synchrone si les IOR des opérateurs reste constant. Un IOR différent de un permet par exemple, de sur ou sous échantillonner un graphe.
La mise en oeuvre d'un IOR différent de un, et ce dans l'optique d'une évaluation statique, met en oeuvre des mécanismes complexes de synchronisation. Les arcs contiennent des tampons permettent la synchronisation des deux opérateurs.
On peut imaginer un opérateur de sommation qui nécessite un certain nombre de valeur en entré pour produire une valeur en sortie.

Figure 16 : opérateur `if` synchrone.

Horloge

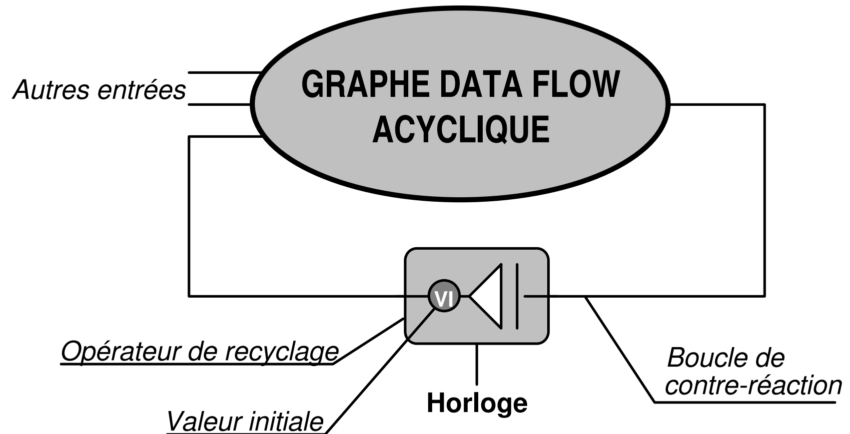
Il faut entendre par impulsion présence d'un jeton.

La synchronisation de la circulation de tous les jetons peut être obtenue par une entrée spéciale dans certains les opérateurs, une entrée d'horloge. A chaque impulsion de cette horloge, les opérateurs consomment leur jeton en entrée, effectuent leur calcul, et place le résultat sur leur sortie.
L'horloge peut être vue comme un flot de contrôle, externe (indépendante du graphe Data Flow proprement dit), et globale (à un graphe). La présence de cette horloge permet de d'apporter les mêmes avantages que des IOR différents de un, à savoir une synchronisation adaptée, sans en avoir la complexité.
L'horloge peut s'apparenter au compteur ordinal des microprocesseurs, lequel joue le rôle du chef d'orchestre du programme. Ici le chef d'orchestre est global et agit d'une manière diffuse.

Opérateur de recyclage

Mais l'adjonction d'une horloge est-elle justifiée sur tous les arcs du graphe. En effet, seules les contre-réactions sont directement concernées car ce sont elles qui pose problème. Est-il dans ce cadre utile d'ajouter à tous les opérateurs la prise en compte de l'horloge?
En automatisme, la contre-réaction est couramment utilisée et parfaitement modélisée. Il s'agit d'un opérateur spécial dont le comportement n'est pas dicté par les mêmes règles que les autres opérateurs.
Cet opérateur effectue le recyclage des données "de la sortie vers l'entrée". Il possède une valeur initiale.
On peut schématiser cet opérateur de la manière suivante :

Figure 17 : opérateur de recyclage.



Le graphe Data Flow ci-dessus est un graphe acyclique. L'opérateur de recyclage est placé pour effectuer une contre-réaction. Sa commande, l'horloge est externe au graphe. Au top horloge, il recopie la valeur de son entrée (à droite) sur sa sortie. La valeur disponible alors sur sa sortie a le même comportement qu'une constante, et fourni la même valeur indéfiniment, jusqu'au prochain top horloge.
Avec cet opérateur, tous les problèmes Data Flow sont résolus : il n'y a plus de graphe cyclique, la contre-réaction est autorisée moyennant l'utilisation d'un opérateur spécial, et enfin le graphe lui-même est simplifié et sa théorie connue et maîtrisée.
Il subsiste cependant une question demeure en suspend : de quelle manière l'horloge est-elle pilotée, et qui en est responsable ?
On peut remarquer que cette horloge agit comme un flot de contrôle vis-à-vis du graphe Data Flow. Avec cet opérateur, beaucoup de choses sont possibles. Toutes les entrées horloge peuvent être connectées entre elles et à une horloge commune, le graphe est alors synchrone. Mais il est possible aussi de disposer de plusieurs horloges ayant des fréquences différentes. Dans ce cas le comportement est alors à étudier.

CONCLUSION

L'étude des graphes Data Flow au cours de cette année a apporté bien plus de questions que de réponses. Mais les travaux les concernant sont récents et les modèles utilisés par les uns et les autres divergent encore beaucoup. Le modèle adopté par Lee et Messerschmitt [2] utilise un IOR différent de un. Cette conception oblige le concepteur à placer des tampons sur la plupart des arcs. L'étude de la synchronisation s'en trouve considérablement compliquée. Cependant, nous pensons qu'il y a moyen de parvenir au même résultat de manière beaucoup plus simple.
Les travaux de Sharp [16] nous semblent intéressants, mais nous pensons que l'utilisation de certains de ses opérateurs Data Flow est à proscrire car elle désynchronise le graphe. De plus il rejette l'utilisation des cycles dans les graphes, interdisant toute contre-réaction. Or concevoir une application de traitement du signal ou d'automatisme sans contre-réaction nous semble une tâche bien difficile, sinon impossible.
De notre étude, nous avons dégagé plusieurs points important :

- le mode *data-driven* et le mode *demand-driven* sont très similaires. Chacun introduit des avantages et des inconvénients, et la véritable question est ailleurs,
- L'IOR doit être égal à un, les désyncronisation, si tant est quelles soient indispensables, devront se faire par un autre moyen,
- l'utilisation d'une horloge est sans doute indispensable,
- l'utilisation d'un opérateur de recyclage des données est sûrement le seul moyen sémantiquement correct d'autoriser les contre-réactions dans un graphe.

Le lecteur aura sans doute noté notre prudence dans nos affirmations. Ceci est dû au fait que notre étude des graphes Data Flow est partielle. Il reste à approfondir les points soulevés.
Mais rappelons que l'objectif n'est pas de fournir une implémentation des graphes Data Flow, mais un environnement de programmation multi-DSP. A l'heure qu'il est, nous ne sommes pas du tout certain que l'utilisation des graphes Data Flow est un bon moyen pour parvenir à nos fins.
L'une des démarches possibles pourrait être très différente de celle-ci. Afin de fournir un environnement de programmation "graphique", nous pourrions partir des langages fonctionnels, sous leur forme textuelle, puis fournir un "utilitaire" graphique capable de traduire certains aspects du langage sous forme de schéma bloc. La base serait alors les langages fonctionnels, donc le λ-calcul, et non les graphes Data Flow, réduits à leurs aspects graphiques.

PARTIE 3: INTERPRETEUR & COMPILATEUR SCHEME

Nous voulons utiliser le λ-calcul comme langage intermédiaire pour traduire un graphe de spécification d'une application. Le λ-calcul reposant sur une théorie solide nous permet de minimiser le risque d'erreur due aux langages impératifs.
De plus le λ-calcul est fonctionnel, ce qui permet d'extraire le parallélisme sous-jacent d'une application d'une manière beaucoup plus efficace. En effet, un ingénieur qui décrit une application à l'aide d'un langage impératif détruit le parallélisme pour en faire une séquence d'instruction. Il est alors plus difficile de retrouver ce parallélisme.
Scheme est un langage fonctionnel basé sur Lisp. Il est construit de manière à éliminer la plupart des caractéristiques des langages impératifs. De plus il est entièrement "*λ-définissable*" et offre la particularité d'être aisément compilable.
Nous disposons d'un Interpréteur Scheme, *scm*, écrit Aubrey Jaffer [5]. Cet interpréteur fonctionne parfaitement, est disponible en code source sous plusieurs environnements. Il nous à parmi de découvrir ce langage et sa puissance expressive. Nous avons avec lui testé différentes implémentations graphe Data flow, comme nous l'expliquons plus loin.
Nous sommes convaincus qu'une version dégradée de Scheme, ne conservant de lui que les mécanismes essentiels, jouera parfaitement le rôle d'un interpréteur de λ-expressions. Pour pouvoir effectuer ce travail, nous devrons forcément modifier l'interpréteur.
Or l'implémentation de *scm* ainsi que les choix structuraux qui ont été fait nous ont conduits à penser qu'il serait beaucoup plus judicieux d'écrire notre propre interpréteur, en utilisant des techniques proches de celles utilisées dans les compilateurs. C'est donc pour cers raisons que nous avons bâti un interpréteur Scheme, nommé *gsm*. A l'heure qu'il est, il implémente la plupart des mécanismes prescrits dans la révision 4 du MIT.
Nous allons dans ce chapitre décrire les choix que nous avons fait et les mécanismes sous-jacents, sans apporter trop de détails d'implémentation qui n'entrent pas dans le cadre de ce rapport.

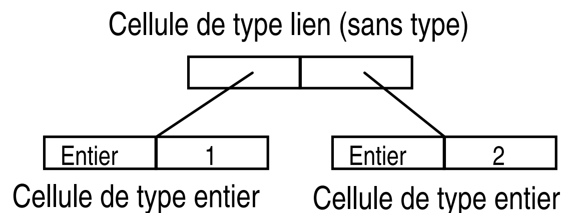
Outils De Base

Nous présentons ici les outils indispensables avant de pouvoir commencer à travailler sur l'interpréteur lui-même. De ces outils de base dépendent en grande partie l'efficacité finale de l'interpréteur. Parmi ces outils, il y a la représentation des données dans l'interpréteur, qui est très liée à la gestion de la mémoire, et la gestion de la table des symboles ou Scheme entrepose toutes les variables créées.

Atomes

Les atomes sont les structures de données terminales possédant une valeur et un type. L'on rencontre par exemple l'atome de type entier et de valeur 21. Scheme définit une collection de types possibles pour les atomes. Nous avons par exemple les types EXACT, INEXTACT, qui sont des nombres entiers ou réels, STRING pour les chaînes de caractères, CHARACTER représente les caractères, etc.
Il y a aussi des atomes qui n'ont pas de type (et de ce fait n'en sont plus). Ceux-ci sont des cellules qui lient d'autres cellules ou atomes entre eux. Par exemple la paire (1, 2) est représentée par :

Figure 18: type de cellules



On peut aussi trouver un atome de type STRING :

Figure 19: type de mémoire
Les différents types de mémoire sont abordés plus bas.

Les cellules sont donc polymorphes et contiennent tantôt des couples type valeurs, tantôt des couples de liens. Mais toutes ont une structure à deux champs, tête et queue, ou **CAR** et **CDR**. Ce sont ces champs qui doivent pouvoir s'adapter aux valeur qu'ils contiennent.
Dans notre implémentation, les champs CAR et CDR des cellules sont des unions au sens du langage c entre les différents types de données qu'ils seront amenés à contenir.

```
/* the cell structure */
typedef struct s_CELL {

  /********* THE CAR UNION ****************/
  union {
    GSM  car;                /* generic car */
    WORD len;                /* vector length */
    WORD type;               /* data type */
  } _car;                    /* car union */

  /********* THE CDR UNION ****************/
  union {
    char             c;      /* cdr.char */
    GSM              cdr;    /* generic cdr */
    FUNC             f;      /* cdr.c_function */
    int              i;      /* cdr.int */
    long             l;      /* cdr.long */
    void           * p;      /* cdr.pointer */
#   ifdef __REAL
    real           * r;      /* cdr.real */
#   endif
    char           * s;      /* cdr.string */
    VECTOR           v;      /* cdr.vector */
    struct s_USR   * u;
  } _cdr;                    /* cdr union */

  /********* THE FLAGS ********************/
  WORD garbage  : 1;         /* garbage collector mark */
  WORD immediat : 1;         /* immediat flag */
  WORD code     : 1;         /* code flag */
  WORD vector   : 1;         /* vector flag */
} CELL;
typedef CELL* GSM;
```

Les champs supplémentaires *garbage*, *immédiat*, *code* et *vector* sont des indicateurs à un bit permettant de déterminer le type de la cellule. Ces indicateurs sont exclusifs entre eux mis à part le champ *garbage* utilisé par le mécanisme ramasse miettes.
Avec cette structure, nous obtenons deux résultats très intéressants. D'une part nous avons un encombrement mémoire réduit, et d'autre part nous ne faisons aucune hypothèse quant à la manière dont le compilateur traite les données, surtout en ce qui concerne les pointeurs. Il en résulte une manipulation immédiate des champs, simplement par affectation. Cette structure de donnée particulièrement efficace nous différencie grandement de l'interpréteur *scm*. Ce dernier possède une structure donc les champs CAR et CDR sont des entiers longs non signés. Cette structure est éminemment plus simple que la notre mais impose un certain nombre de manipulation pour déterminer le type des cellules et y ranger des données.

Gestion de la mémoire

La gestion de la mémoire est un gage de qualité d'un interpréteur. Du mécanisme d'allocation de mémoire vive dépend pour une grande part l'efficacité du programme. Les interpréteurs Lisp utilisent en général le mécanisme de ramasse miettes (*garbage-collecting*). Ce mécanisme offre l'avantage de gérer automatiquement la récupération des zones inutilisées de la mémoire. Le programmeur ne se soucie plus de la libération des zones qu'il a allouées, le système s'en occupe pour lui. Toute personne ayant programmé avec un langage "classique", comme pascal ou c, a connu les difficultés de la gestion de la mémoire, au fur et à mesure que le programme grossit.

Garbage signifie poubelle. Tas est utilisé au lieu de heap.

Notre implémentation utilise trois zones mémoires gérées différemment, un *garbage*, un tas et une pile.
Ces trois zones mémoires répondent à des besoins différents. Le *garbage* concerne la gestion des objets Scheme, comme les listes, les vecteurs, etc. Le tas est à rapprocher du tas (ou *heap*) des programmes C ou pascal. Il sert à stoker des objets comme les chaînes de caractères, les tableaux de vecteurs, etc. Enfin la pile est

utilisée pour stocker des adresses de cellules du *garbage* pendant les appels de fonctions.

Le tas

La plupart des bibliothèques c, si ce n'est toutes, offrent une gestion de tas dont les fonctions de manipulation les plus connues sont *malloc()* et *free()*. Mais ces bibliothèques ne donnent pas assez accès aux primitives de la gestion de mémoire, ou si elles le font, c'est sous des formes extrêmement différentes. D'autre part, maîtriser la gestion de la mémoire, c'est maîtriser la principale ressource d'un ordinateur. Ainsi sur un Ibm pc® sous Dos®, on pourra accéder aux zones de mémoire hautes, qu'aucune bibliothèque standard ne sait gérer. Ceci nous conduit à créer notre propre gestionnaire de mémoire.
L'algorithme de gestion du tas n'est pas très compliqué : il est parfaitement décrit sous une forme directement utilisable dans le livre de **Kernighan et Richie [2]**. Cet algorithme est très certainement optimisable (on peut par exemple l'écrire en assembleur pour chaque machine hôte) mais dans le cadre de nos besoins, il suffira amplement.

Ce mécanisme de gestion de la mémoire à deux étages est utilisé dans Windows de Micro Soft.

Mais nous ajoutons une indirection supplémentaire. Notre fonction *gsm_malloc()* ne retourne pas un pointeur directement utilisable, mais un *handle* (qui signifie manche, d'où l'idée de tenir). Pour obtenir un pointeur, il faut appeler la fonction *lock()*. A la fin de l'utilisation du pointeur, on appelle *unlock()* qui libère le pointeur. Ce mécanisme ne sert actuellement à rien (*lock()* et *unlock()* sont des macro-définitions vides). Mais par la suite, il sera possible de gérer directement un tas de mémoire distant, par exemple, ou une zone de remisage sur disque, etc.
Le tas implémenté sait "grossir et rétrécir" selon les besoins du programme. Si le système d'exploitation ne peut plus allouer plus de mémoire, une erreur se produit. De plus nous vérifions qu'à la fin d'une session de travail, tous les objets alloués ont été libérés. Cela permet de détecter des erreurs de programmation.

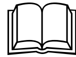
La pile

La pile est la plus simple des trois structures de mémoires mises en oeuvre. Elle est représentée par un tableau d'adresses de cellules (tableau de pointeurs). Elle est manipulée par trois fonctions : *push()*, *pop()* et *popn()*. La première empile un élément dans la pile, la seconde dépile un élément et la troisième dépile n éléments. Notons que la troisième fonction n'est pas essentielle mais qu'elle permet d'augmenter l'efficacité de l'ensemble.
Dans notre implémentation, la pile est allouée dans le tas, ce qui permet d'en régler la taille initiale dynamiquement. Un contrôle de dépassement est effectué.
Mais pour l'instant, la taille de la pile est fixe. Plus tard, nous implémenterons une pile dont la taille s'adaptera aux besoins de l'application.

Le *garbage*

Cette zone de mémoire est utilisée pour stocker les objets Scheme, à savoir les atomes, les listes, etc.
Le langage Lisp tire une grande partie de sa puissance par son mécanisme d'allocation de mémoire. Le programmeur crée des objets, les utilise autant de fois qu'il le désire, mais ne se soucie jamais de libérer l'espace mémoire qu'ils occupent. Cette souplesse se traduit par un accroissement très sensible de productivité par rapport, par exemple au langage c, sur les projets où les structures sont essentiellement dynamiques.

Le garbage est un tableau de cellules. Ce mécanisme repose sur la sémantique même du langage : "tout est liste". En effet un programme Lisp peut être vu comme une grappe de cellules, ou un arbre. Chaque objet crée est "vu" par un autre au-dessous de lui, jusqu'à la racine. La racine est le point d'entrée de ce mécanisme : il commence par marquer toutes les cellules du *garbage* ; puis il parcourt l'arbre en commençant par la racine en démarquant les cellules rencontrées ; enfin il parcourt à nouveau l'ensemble du *garbage* : toutes les cellules marquées sont inutilisées et réintégrées dans la liste des cellules disponible. Si la cellule à réintégrer contient un pointeur vers le tas, comme une cellule de type STRING, alors ce pointeur est libéré avec la fonction *free()*. Une excellente description de cet algorithme est faite dans le livre d'Alfred Aho [1].

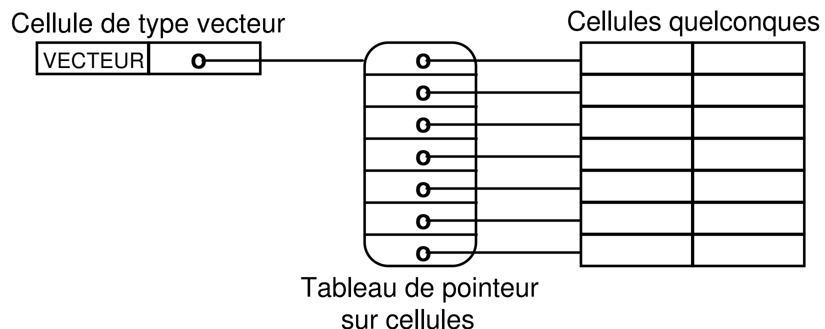


Dans notre implémentation, le *garbage* a une taille fixée au moment du lancement de *gsm* ; on peut ainsi en régler la taille. Une fois alloué, la garbage ne peut pas changer de taille. Dans le future, nous implémenterons un garbage dynamique.

Vecteurs

Le type vecteur n'est pas essentiel dans Scheme. En effet, un vecteur peut être représenté avec une liste, mais il fait partie du noyau dans notre implémentation pour des raisons d'efficacité. Un vecteur est un tableau de cellules de taille fixe. Les cellules font partie de la mémoire du *garbage*, alors que le tableau fait partie du tas. Nous représentons un vecteur comme ceci :

Figure 20 : représentation des vecteurs



Les vecteurs nous seront très utiles pour manipuler efficacement des groupes de cellules, comme, les symboles, qui associent un nom à une valeur.

Table des symboles

La table des symboles est l'élément vital d'un interpréteur. En effet, l'interpréteur agit "en temps réel" sur les données, et il passe son temps à associer des valeurs à des noms de symbole, et à rechercher quelle est la valeur des symboles. Ce va-et-vient est coûteux en temps machine et il convient d'avoir une structure de stockage efficace. Dans Scheme, le problème est atténué par les liaisons statiques aux symboles : l'interpréteur remplace les noms des symboles par l'adresse de la valeur du symbole. A l'exécution, la valeur du symbole n'est plus recherchée dans la table.

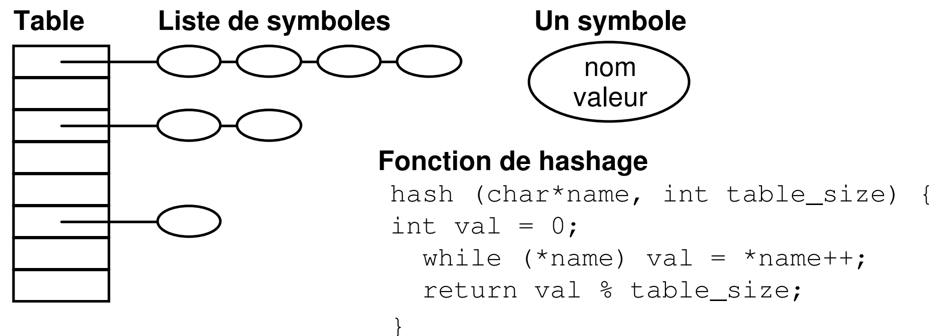
Pour en savoir plus, reportez-vous au livre d'Abelson et Susman [13].

Nous nous sommes inspirés pour la gestion de la table des symboles (*hash table*) du libre d'Alfred Aho, John Hopcroft et Jeffrey Ullman [1]

Une table des symboles et un tableau de taille fixe. Chaque entrée du tableau est une liste de symboles. Un symbole est un couple nom-valeur. Lorsque l'on veut ajouter un symbole à la table, on applique au nom du symbole une fonction dite de *hashage* qui retourne un index dans le tableau. Le symbole est ajouté à la liste des symboles de l'entrée correspondante du tableau.

Dans notre implémentation, la table des symboles est allouée dynamiquement, ce qui permet de choisir sa taille au lancement du programme. La fonction de *hashage* fait la somme de tous les caractères présents dans le nom du symbole modulo la taille de la table.

Figure 21 : table des symboles



Interface avec le langage c

Notre implémentation de Scheme utilise le passage de paramètre du langage c. Cela signifie que les paramètres sont empilés dans la pile du programme. Le contrôle du nombre de paramètre est effectué par la fonction appelante, alors que le contrôle du type des données est effectué par la fonction appelée. La fonction appelante est la fonction Scheme *apply()* qui applique à une fonction des arguments. Les arguments sont évalués avant l'appel.

Appel de fonction

Pour ce faire, nous avons des cellules de type CODE (champs *code* de la structure *CELL*). Une telle cellule contient le nombre d'arguments à passer à la fonction dans le champ CAR. Le champ CDR contient quant à lui un pointeur vers une fonction c.
L'usage de la pile c pour passer les paramètres possède un inconvénient. S'il n'y a plus de place dans le *garbage*, une récupération des cellules libres risque être effectuée (*garbage-collecting*). Les cellules présentent dans la pile, si elles ne sont rattachées à aucun objet, vont être considérées comme inutilisées et libérées. Au retour du *garbage-collecting*, certains objets Scheme risquent d'avoir été réintégrés dans la liste des cellules disponibles et donc de n'être plus valides.

Pile

Il est donc indispensable d'avoir un mécanisme de pile annexe. Le mécanisme ramasse miettes parcourra aussi cette pile.
Mais alors quel est l'avantage de passer les paramètres dans la pile du c, s'il faut aussi les empiler dans une pile annexe ? Nous avons vu que les structures de Scheme sont très souvent arborescentes. Il n'est donc nécessaire d'empiler dans la pile annexe que la racine de l'arbre, et non la totalité des feuilles. Ainsi nous protégeons tout l'arbre contre un *garbage-collecting* et nous utilisons la pile du c qui utilise des instructions machine et qui est donc extrêmement optimisée.
La stratégie adoptée est la suivante :

- ❑ les objets passés en paramètres des fonctions c seront empilés (la fonction n'aura donc pas à le faire),

- les objets créés à l'intérieur de la fonction devront être empilés si celle-ci contient un appel qui "risque de provoquer une récupération". En règle générale, tout objet créé dans une fonction sera empilé à sa création et dépilé au retour de la fonction.

Par exemple nous voulons écrire la fonction factorielle en c. Nous avons :

```
GSM factorielle (GSM n) {
GSM l;

  /* Contrôle des arguments : n est un entier */
  if (! IsExact (n))
    error("arguments invalides", GOTO_TOPLEVEL);

  /* si n vaut 0 ou 1 (GINT retourne la valeur entière
     contenues dans une cellule */
  if (GINT (n) == 0 || GINT(n) == 1)
    return make_atom (INTEGER, 1);
  /* mul_2 multiplie deux nombres entre eux */

  /* on empile les résultats intermédiaires comme n-1 */

  l = mul (n, PUSH (factorielle (PUSH (sub (n, 1)))));

  POP();

  POP();

  return l;

}
```

Cette fonction montre le mécanisme de pile. Les résultats intermédiaires comme le calcul de (n-1) ou de factorielle (n-1) peuvent être perdus car ils ne sont rattachés à aucune structure (liste, arbre, etc.). Ils doivent donc être empilés pour se prémunir d'un *garbage-collecting*.

Type de fonction

Nous n'entrerons pas ici dans les détails de l'implémentation. Mais il est important de dire qu'il existe plusieurs sorte de fonction Scheme.

Dans la présentation du langage, nous avons pu remarquer que certaines fonctions, comme `define`, ne se comportaient pas comme les autres. En effet `define` doit être appelée sans que ses arguments soient évalués. Cette différence de comportement se traduit aussi dans l'implémentation. Il est donc nécessaire de faire une distinction entre une fonction quelconque qui demande l'évaluation de ces arguments, et les formes spéciales qui dans la plupart des cas attendent des arguments non évalués.

INTERPRETATION & COMPILATION

Nous présentons ici le mécanisme d'interprétation. Ce mécanisme a pour charge, d'une part, de construire une expression à évaluer à partir d'un flot de caractères (clavier, fichier), c'est l'analyse, et d'autre part à évaluer cette expression et d'afficher sont résultat, c'est l'évaluation.

En ce qui concerne l'analyse, nous nous sommes inspirés du livre de A. Aho, R. Sethi et J. Ullman [2] qui présente les techniques indispensables à l'élaboration d'un compilateur. En première partie de ce livre on trouve un analyseur syntaxique dont nous avons repris et adapté les principes de fonctionnement.

Analyse syntaxique

L'analyse syntaxique est effectuée sur un flot de caractères représentant l'entrée du programme. Cette entrée peut être le clavier où l'utilisateur tape ses commandes à l'interpréteur, ou d'un fichier texte dirigé comme on pourrait le faire sous Unix avec la commande :

```
gsm < fichier_programme.s
```

L'analyse syntaxique dans gsm se compose de deux modules distincts.

Le premier est l'analyseur de lexèmes. Les lexèmes sont un groupement de caractères ayant une valeur reconnue par l'interpréteur. Par exemple 123 est le lexème entier▫123, "abc" est le lexème chaîne▫{a, b, c} fonction est l'identificateur▫ fonction. Par rapport à d'autres langages, il y a peu de lexèmes dans Scheme, ce qui en fait un langage particulièrement simple. L'analyseur de lexèmes se comporte comme une machine à états finis.

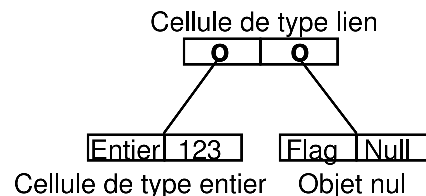
Le second module est le constructeur d'expression. Dans *gsm*, une expression est représentée sous la forme d'une liste dont chaque élément est un lexème. Dans cette phase, *gsm* n'a pas fait la moindre évaluation, il s'est contenté de construire une liste de lexèmes.

Par exemple si on tape au clavier

```
⇨123
```

gsm construit la liste :

Figure 22 : représentation de l'expression 123.



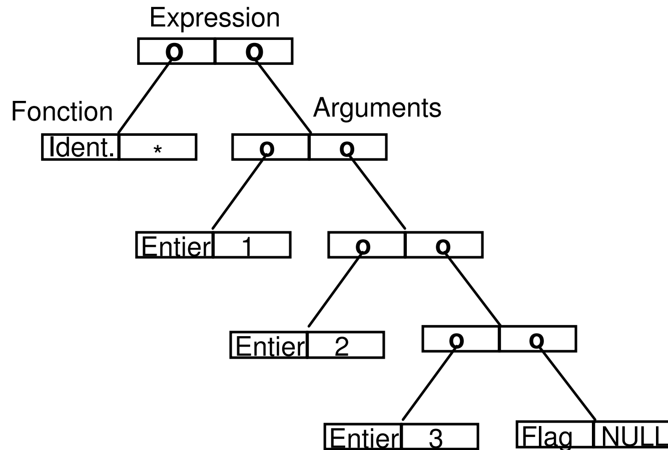
Cette liste va ensuite être donnée à l'évaluateur qui va évaluer chaque terme de la liste.

Pour un appel de fonction cela donne :

⇨ `(* 1 2 3)`

qui se traduit en :

Figure 23 : représentation de (* 123)



On peut représenter à l'aide ce ces listes toutes les expressions possibles. Ces expressions sont maintenant données à l'évaluateur.

Evaluateur

L'évaluateur évalue les expressions retournées par l'analyseur syntaxique.

Si ces expressions ne sont pas des listes l'évaluateur interprète l'expression directement. Si l'expression est un identificateur, l'évaluateur recherche dans la table des symboles la valeur de cet identificateur. Si ce dernier n'est rattaché à aucun symbole, il se produit une erreur.

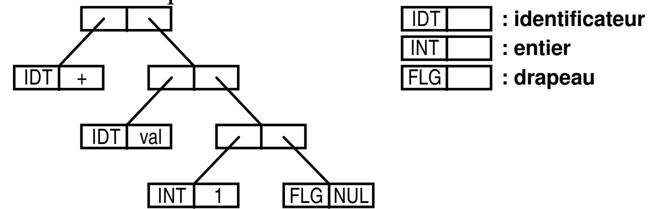
Si l'expression est une liste, l'évaluateur la considère comme un appel de fonction. Il évalue alors le premier élément de la liste et appelle la fonction apply. Ce premier élément doit être une cellule de type CODE (autrement dit, la valeur attachée au premier élément de la liste doit être une fonction). La fonction `apply()` reçoit comme premier paramètre une cellule de type CODE et une liste d'argument. Ces arguments ne sont pas évalués.

La fonction `apply()` vérifie que le premier argument est du type CODE. `apply()` vérifie ensuite que le nombre d'objet dans la liste correspond au nombre d'arguments attendus par la fonction appelée. Selon le type de fonction, les arguments sont évalués (rappelons que certaines formes spéciales n'attendent pas d'arguments évalués). `apply()` appelle ensuite la fonction c correspondante au premier paramètre avec les arguments évalués ou non. C'est le résultat de cet appel qui est retourné.

On constate que selon ce schéma, n'importe quelle expression peut être évaluée (produisant ou non une erreur).

Nous montrons le mécanisme de l'expressions `(+ val 1)`.

1°) acquisition de la chîne de caractères.
    buffer <- (+ val 1)
2°) segmentation de cette chaîne en unité syntexiques
    '(', '+', 'val', '1', ')'
3°) fabrication de l'expression à évaluer



4°) evaluation : eval (exp)
    a°) Si exp est entier, retourne exp
    b°) si exp est un identificateur
        chercher dans l'environnement courant exp
        si trouvé, retourne sa valeur
        sinon erreur.
    c°) si exp est une liste
        retourne apply (eval (CAR(exp), CDR(exp))
5°) application : apply (func, arg)
    a°) si func n'est pas une fonction, erreur
    b°) si func n'est pas une forme spéciale, évaluer tous les arguments
        while (arg != NUL)
            CAR(arg) = eval (CAR(arg))
    c°) appel à la fonction
        retourne call_proc (func, arg)

Compilation

La compilation d'expression dans Scheme est rendue possible par la liaison statique des données du langage. Cette compilation ne s'effectue que sur les λ-expressions et leur formes dérivées. L'avantage de cette technique est un gain de temps à l'exécution considérable car la valeur des symboles n'est plus recherchée dans la table.

Lambda

La compilation d'une λ-expressions fournit une structure de données reconnue en tant que telle est évaluée dans l'évaluateur par une fonction spéciale, l'évaluateur de λ-expressions compilées.
Une λ-expression peut être considérée comme un monde fermé. Ce monde fermé est l'environnement attaché à l'expression au moment de sa compilation. Toute opération sur les symboles passe par cet environnement. A la fin de la compilation, cet environnement est détruit. Nous prendrons comme exemple la λ-expressions :

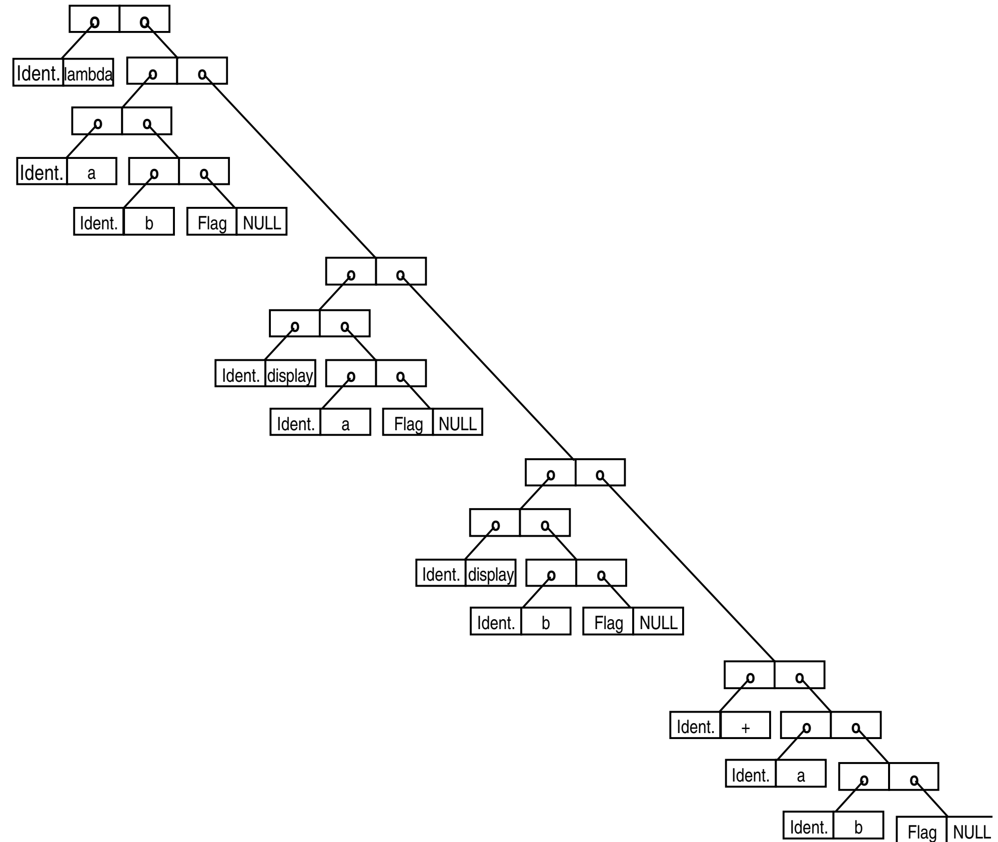
```
(lambda (a b)
  (display a)
  (display b)
  (+ a b))
```

Structure de donnée des λ-expressions

Cette structure est basée sur un vecteur. Ce vecteur a pour dimension le nombre d'argument formel augmenté de un, pour la liste d'instruction.

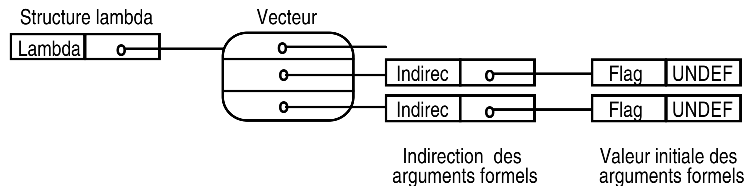
A la sortie de l'analyse syntaxique, nous avons la structure :

Figure 24 : analyse d'une λ-expression



La première phase de la compilation constitue à créer une structure de donnée :

Figure 25 : structure de donnée vierge de λ-expression



a

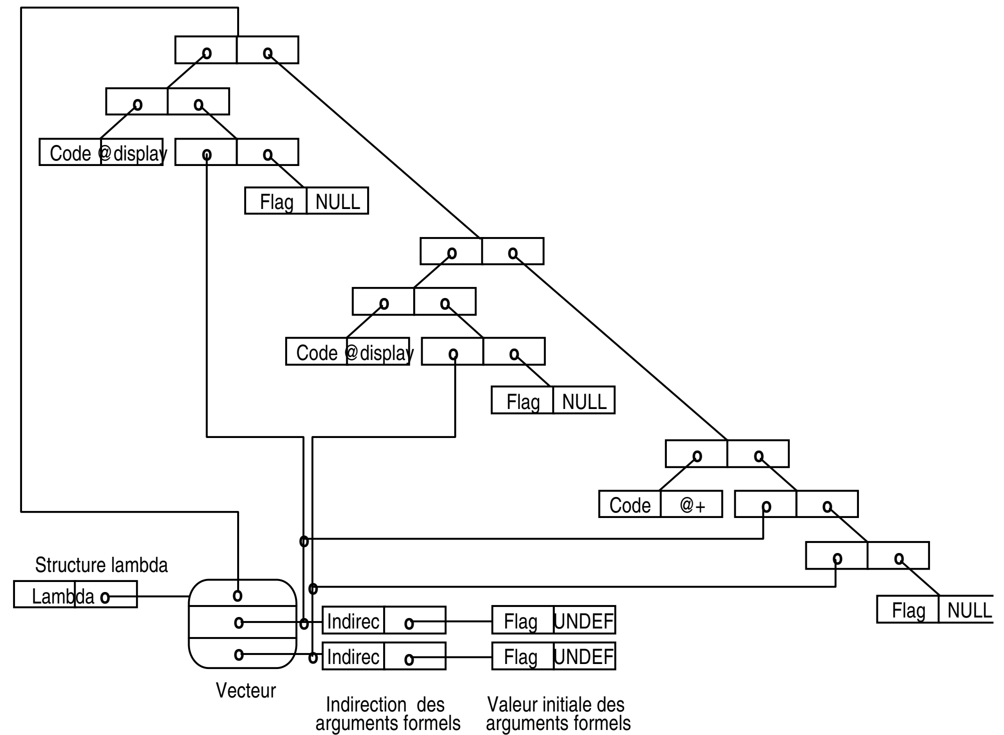
Cette structure de donnée se présente sous la forme d'un vecteur, cependant, le type de la cellule de tête est LAMBDA. Le premier champ de la structure correspond à la liste d'instruction contenue dans le corps de la λ-expression ; pour l'instant, cette liste n'est pas encore compilée. Les champs suivants correspondent aux arguments formels. La valeur de ces arguments formels est obtenue par indirection. Parallèlement, les symboles associés aux arguments formels sont créés dans un environnement temporaire, dont environnement parent est environnement global. Ces symboles sont créés avec comme valeur l'indirection ci-dessus mentionnée. Ces indirection pointent pour l'instant sur un indicateur UNDEFINED.

Compilation

La deuxième phase est la compilation proprement dite du corps de la λ-expression. L'idée de base de cette compilation est de remplacer dans le corps tous les identificateur par leur valeur. En ce qui concerne les arguments formels, leur valeur

est une indirection. Tous les symboles non trouvés, ni dans environnement temporaire, ni dans environnement global sont créés dans ce derniers avec comme valeur un indicateur UNDEFINED. Ce point permet d'écrire un programme avec des références en avant ou *forward-reference*.

Figure 26 : compilation d'une λ-expression



Dans la liste d'instruction, tous les symboles ont disparu, remplacés par leur valeur. Pour évaluer cette λ-expression, il suffit maintenant de remplacer les valeurs pointées par les indirection par les valeurs des arguments formels et de donner la structure à l'évaluateur.

Let

Dans l'introduction à Scheme, nous avons pu remarquer que l'instruction `let` pouvait être facilement remplacée par un `lambda`, comme nous le montrons sur cet exemple :

```
(Let ((a 3)(b 4)) (display a)(+ a b))
```

⇔

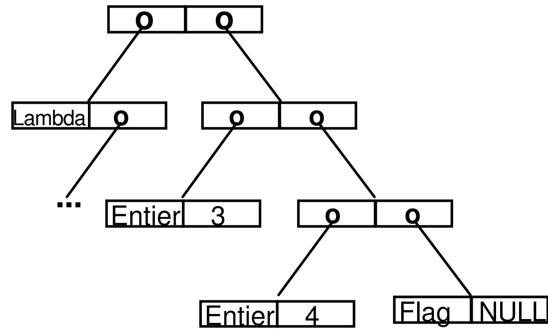
```
((lambda (a b) (display a)(+ a b)) 3 4)
```

Mais il faut se rendre compte qu'une expression `let` n'est pas simplement un résultat, en l'occurrence, le résultat de la dernière expression du corps de l'expression. En effet, dans l'exemple précédent, le résultat de l'expression est `7`. Mais si on écrit :

```
(define (foo)
  (Let ((a 3)(b 4))
  (display a)(+ a b)))
```

La valeur de `foo` n'est pas `7`, mais le résultat de la compilation. Lorsque l'on appelle `(foo)` on doit avoir affiché `37`, le `3` provenant du `(display a)` et le `7` du `(+ a b)`.
Une autre difficulté vient du fait qu'un `let` se traduit par une application. Dans l'exemple, c'est l'application de `3` et de `4` au `lambda`. Cela se traduit par :

Figure 27 : compilation d'un let;



Cette structure est l'application d'une fonction à ses arguments. Or pour l'évaluateur, un appel à une fonction retourne une valeur, et non une application. L'évaluateur doit donc distinguer cette structure ; si on se trouve dans environnement global, le résultat d'un let et l'évaluation du résultat de la compilation, sinon le résultat est le résultat de la compilation.

EXTENSION DE SCHEME

Fichiers compilés

Il est souvent utile de cacher les modalités de fonctionnement d'un programme à son utilisateur. Cela n'est possible qu'en codant les programmes manière à les rendre illisibles.
La compilation en général s'acquitte très bien de cette tâche, car les fichiers compilés sont illisibles. De plus ils sont dégradés, c'est à dire qu'ils ont perdu de l'information, comme le nom des variables. La compilation a pour objet d'accélérer l'exécution d'un programme en le rendant plus facile à lire pour la machine.
Une extension indispensable est de fournir la possibilité de compiler les programmes. Le résultat de cette compilation pourrait être un programme exécutable, dans le code de la machine. Mais cela est beaucoup trop complexe dans le cadre de nos besoins.
Une autre voie possible est la sauvegarde dans un fichier de l'image du tas est du garbage. Le chargement d'un programme serait donc simplement la lecture d'un tel fichier dans la mémoire. De plus ce procédé nous donne à peu de frais la possibilité de fournir des programmes exécutables binaires : il est possible de recopier cette image dans un tableau statique d'un chargeur. Lors de l'exécution du chargeur, celui-ci évalue son tableau statique, qui est devenu un programme Scheme compilé.

Librairies à liens dynamiques

Les librairies à liens dynamiques sont un moyen très souple et très sécurisant d'ajouter des fonctionnalités à un programme sans le recompiler. Il permet aussi de configurer dynamiquement le programme en question, en chargeant par exemple un pilote d'imprimante; Cela ne peut se faire au moment de la compilation, à moins d'inclure tous les pilotes des imprimantes que l'on désire contrôler, et celles qui n'existent pas encore aussi!
Dans l'environnement Windows, la gestion des bibliothèques est fournie par le système. Cette gestion est si souple et si transparente pour l'utilisateur que Windows lui même est constitué d'un ensemble de bibliothèques dynamiques.
Dans l'environnement Dos, la tâche fut plus ardue. En effet, comment fournir un mécanisme aussi dynamique dans un environnement préhistorique. Nous y sommes parvenus à grand frais. Les bibliothèques sont des programmes résidents préchargés en mémoire et pilotés par interruption. Chaque bibliothèque exporte une table de ses fonctionnalité vers le serveur. Le résultat final est assez proche de celui disponible sous Windows, en nous n'en sommes pas peu fière!
Dans l'environnement Unix les choses restent à faire, mais nous ne nous faisons pas trop de soucis. En effet Unix est un environnement multitâche et la communication

entre processus est depuis longtemps utilisée et décrite. Par contre nous aurons le choix des moyens. Il nous faut une fonction qui soit commune au plus grand nombre d'Unix possible. Nous pensons à première vue aux *pipes*, aux signaux, ou aux *sockets*. Affaire à suivre!

Une fois les bibliothèques disponibles, nous pourrons réduire le noyau Scheme à sa plus simple expression, ne gardant de lui que l'essentiel (au sens sémantique). L'interpréteur proprement dit sera construit dynamiquement à coup de bibliothèques dynamiques. Ainsi, simplement en modifiant le nom de telle ou telle bibliothèque dans un fichier texte, nous pourrons changer l'interpréteur Scheme en interpréteur Lucid, ou passer du compilateur pour le DSP TI TMS320C25 à un compilateur pour un DSP Motorola!

SYNTHESE

Le lecteur aura remarqué que ce rapport est constitué de briques, apparemment sans lien entre elles. Cependant, au fils des commentaires dans la marge et de remarques, on s'aperçoit qu'un tissu commun les lie. Ce lien est la clef de voûte du rapport, présentée au début dans le chapitre intitulé Objectifs.
L'objectif est de fournir un environnement de programmation multi-DSP. Cet environnement de programmation se doit de répondre à certains critères de qualité et de fournir un plus dans la conception des applications.
Ce plus est la simplification de la phase de conception et l'assouplissement de la réalisation. Cet outil doit apporter une rupture nette avec le schéma classique : conception-réalisation-déboggage, où le déboggage prend plus de temps que les deux premières étapes réunies.
Une telle rupture ne peut se produire en modifiant radicalement les manières de penser "application". Le fondement théorique tient une place prépondérante dans la conception de l'outil. Or il se trouve qu'il existe une théorie solide en matière de programmation, le λ-calcul, qui ouvre la porte de tous les langages fonctionnels. Mais pourquoi nous intéressons-nous tant aux langages fonctionnels?
Les langages de programmation "classiques", dits impératifs, se heurtent maintenant à leur propre limite. Cette limite se trouve en fait être la limite de l'architecture de la machine sur et pour laquelle ils ont été conçus, l'architecture de Von-Neumann. Cette architecture est essentiellement séquentielle. Elle oblige à avoir une vision très localisé du traitement de l'information. Or les applications modernes sont énormes, et agissent sur des flux de données importants.
La solution actuelle à ce problème est un certain nombre de déclinaisons de cette architecture de Von Neumann. Mais ce soucis de réutilisabilité de l'existant ne doit pas nous faire oublier que si les fondations sont instables, la pyramide l'est aussi.
Les langages fonctionnels s'appuient indirectement, via le λ-calcul, sur les mathématiques, qui sont bien sûr indépendantes de toutes architecture! L'outil mathématique repose sur le concept de preuve, de démonstration. Apporter la preuve d'un programme informatique, c'est construire des fondations stables.
Ainsi notre outil utilisera un langage fonctionnel. Mais lequel? Ce choix est si important pour les langages impératifs, qu'il peut paraître inconscient d'en choisir un maintenant.
Le lecteur aura remarqué par son expérience et à la lumière de notre présentation sommaire de quelques langages fonctionnels, qu'ils se ressemblent tous. Les différences sont surtout dans la syntaxe, qui favorise tel ou tel aspect du langage et des données sous-jacentes, ou dans le mode d'évaluation. Mais ils ont tous le même support, et pour cause : ils sont tous des dérivés du λ-calcul. Alors le choix d'un

langage n'est plus aussi crucial qu'il, n'y paraissait au premier abord. Il nous en faut un qui soit simple, sémantiquement propre, extensible, implémentable, connu, et disponible. Disponible, Scheme l'est ! Son noyau de base est très proche du λ-calcul, c'est un langage connu, vivant, documenté. Le MIT s'en sert comme langage d'apprentissage. Il n'en fallait pas plus pour arrêter notre choix.
L'implémentation de Scheme dont nous disposons fonctionne parfaitement. Cependant les choix d'implémentation fait par son concepteur nous paraissent parfois étranges, et surtout illisible! De plus, il nous faut maîtriser ce langage dans ces moindres détails pour en réaliser un compilateur pour DSP. Nous avons donc entrepris la réalisation de notre propre interpréteur Scheme, *gsm*. A l'heure qu'il est, cet interpréteur met en oeuvre la plupart des recommandations du MIT. Un prochain travail sera de la terminer.
Pour en revenir aux applications modernes, nous avons vu qu'elles traitent des flots d'information importants. Or il existe depuis plusieurs années une branche de l'informatique qui s'intéresse plus particulièrement aux flots de données et qui tente d'en faire une théorie. Ses auteurs s'appuient sur une représentation graphique des applications, sous forme de graphes : ce sont les graphes Data Flow. Jeune, cette théorie est encore incertaine. Elle séduisante dans le cadre de nos objectifs, car les mots qui lui sont assortis sont graphe, graphique, parallélisme, flot de données, etc. Cependant, les incertitudes soulevées ne permettent pas d'affirmer que nous les utiliserons dans notre outil. Il se peut que nous choisissions une représentation graphique ressemblant aux graphes Data Flow des langages fonctionnels. Cette représentation sera certainement limitative, et toutes les formes du langage (du moins celle dont nous aurons l'utilité) ne pourront être représenté. Il reste bien sûr à détecter ces cas "pathologiques".
C'est donc ces trois domaines, le λ-calcul, les langages fonctionnels et les graphes Data Flow, que nous nous sommes attachés à étudier. La réalisation "pratique" ce cette étude est la réalisation d'un interpréteur Scheme.
Etude théorique, puis réalisation, il nous semble que la démarche est la bonne et correspond au meilleur moyen d'atteindre les objectifs dictés par le projet de recherche. Bien sûr il s'agit là d'une étude préliminaire. Tous les domaines abordés doivent être approfondis.

Sur un plan plus personnel, et dans ce cadre j'utiliserais la première personne, la mieux placée pour décrire mes impressions, ce stage s'inscrit dans la formation en vue de l'obtention du D.E.A. T.T.I de L'Université de Nice-Sophia Antipolis. C'est un stage d'initiation à la recherche qui doit être un résumé de la vie de chercheur. Il se déroule dans cette optique dans un laboratoire de recherche, en l'occurrence, le Laboratoire I3S.
Sur le plan de la recherche, et donc du projet, j'ai été passionné par ce projet. Vaste, il touche à beaucoup de domaines de l'informatique, certains parfaitement connus, d'autres à l'état d'ébauches. Le travail bibliographique m'a permis de découvrir bon nombre d'aspect dont je ne soupçonnais pas même l'existence. Ce foisonnement, cette excitation des idées est l'un des plus stimulant et des plus enrichissant. J'ai le sentiment d'avoir beaucoup progressé au cours de cette année de stage.
Sur le plan relationnel, je rends hommage à mes directeurs de stage, MM Boéri et Demartini, sans qui ce cheminement n'eut été possible. Ils ont su me piloter constamment. Nos discussions à battons rompus ont été passionnantes et bon nombre des idées de ce rapport leur sont dues. Ils ont su aussi prendre en compte mes initiatives en me laissant ainsi un degré de liberté que j'ai apprécié.
Je remercie aussi tous les membres du laboratoire. Je ne peux les citer ici ayant déjà dépassé mon quota de pages. Ils m'ont accueillis à bras ouverts et ont été si souvent disponibles pour répondre à mes interrogations. Grâce à eux, nous (les DEA) nous sommes sentis faire partie à part entière du laboratoire.

Ainsi, au terme de cette expérience, j'affirme que la vie de chercheur est celle qui me convient, et j'espère que j'aurai la chance d'y parvenir.

BIBLIOGRAPHIE

[1] **A.Aho, J.Hopcroft, J.Ullman**
Data Structures and Algorithms
Addison-Wesley Publishing Company, Inc & InterEdition

[2] **A.Aho, R.Sethi, J.Ullman**
Compilers
Addison-Wesley Publishing Company & InterEdition

[3] **A.Faustini**
pLucid Interpreter
Departement of Computer Science, Arizone State University, Tempe, Arizona 85297, USA

[4] **AE.Lee, H.Wai-Hung, EE.Goei, JC.Bier, S.Bhattacharyya**
Gabriel : A Design Environment for DSP
IEEE Trans on Acousyics, Speech and Signal processing Vol. 37, n°11, nov.86

[5] **Aubrey Jaffer**
SCM : A Scheme Implementation 4.00
Ethernet :jaffer@ai.mit.edu
Aubrey Jaffer, 84 Pleasant St., Wakefield MA 01880

[6] **B.W Kernigham, D.M.Ritchie**
The C Programing Language
Prentice-Hall, Masson

[7] **Etude et réalisation d'un système d'aide à la conception d'architectures multiprocesseurs (SAM-1) n° DRET 87/235**
Contrat DRET/LASSY/SIMULOG, Mars 1988 à Décembre 1989.

[8] **Etudes sur la conception Assistée de Processeurs Synchrones Spécialisés (CAPSYS-1) n° DRET 89/240**
Contrat DRET/LASSY, Mai 1990 à Mai 1992

[9] **G.Durrieux, K.Kessaci, M.Lemaître**
Spécification et synthèse de circuits électroniques digitaux
Convention DRET n°89/002 - ONERA/CERT/DERI

[10] **G.Durrieux, K.Kessaci, M.Lemaître**
Transe : An experimental Design Tool
Conference on Algoritms and parallel VLSI Architecture II - Château de Bonas - France - June 91

[11] **G.Menez, M.Auguin, F.Boéri, C.Carrière**
Contribution of compilation techniques to the synthesis of VLSI architectures
"Working Conference on Architecture and Compilation Techniques for Fine and Medium Grain Parallelism" - Jan 93 - Orlando, Florida, USA

[12] **G.Revesz**
Lambda-Calculus, Combinators, and Functional Programming
Cambridge Tracts in Theoretical Computer Science 4

[13] **H. Abelson, GJ.Sussman, J.Sussman**
Structure and interpretation of Computers Programes.
MIT PRESS & INTER EDITION

[14] **H.Glaser, C.Hankin, D.Till**
Principles of Functional Programming
Prentice/Hall

[15] **J.Demartini, A.Ferro, C.Templier**
Luxifere : a tool for functionnal modeling and simulators generating
International Symposium on Mathematical and Intelligent Models in System Simulation - IMAGS - Sept 90 - Bruxelle

[16] **J.A.SHARP**
Data Flow Computing
Ellis Horwood Serie COMPUTER AND THEIR APPLICATIONS

[17] **P.J. Landin**
The Next 700 Programming Langages
Communications of the ACM, March, 1966

[18] **Simulation d'architecture multiprocesseur (SAM-2) n°DRET 91/383**
Contrat DRET/LASSY/SIMULOG, Décembre 91 à Décembre 92

[19] **SL. Peyton Jones**
The implementation of Functional Programming Languages
Prentice Hall International, Hemel Hempstead, Hertfordshire - UK

[20] **Synthèse automatique de processeur synchrone spécialisé (CAPSYS-2) n°DRET 92-34-060-00-470-75-01**
Contrat DRET/LASSY, Octobre 92 à Octobre 93

[21] **W.Clinger & J.Rees (Editors)**
Revised^4 report on the Algorithmic Langage SCHEME
Réseau Ethernet : @

[22] WW.Wadge, EA.Ashcroft
Lucid, the DataFlow Programming Language
ACADEMIC PRESS

INDEX

affectation, 58
allocation de mémoire, 68
alpha-conversion, 29
analyse syntaxique, 72
application, 25; 70
arbitrage des accès, 18
arbre abstrait, 12
argument, 44
argument formel, 49; 75
atomes, 66
béta-conversion, 27
béta-réduction, 28
bottom, 42
Capsys, 10
capture des noms de variables, 31
car, *46*; 49; 66
caractère strict, 39
cdr, *46*; 49; 66
cellule, 67
champ données, 17
champ entrées/sorties, 17
champ programme, 17
clause where, 42
combinateur de point fixe, 34
compilation, 20; 73
comportement opérationnel, 38
compteur ordinal, 58
cons, *46*; 49
contre-réaction, 60; 63
corps de fonction, 26
curryfication, 25
D.S.P., 16
Data Flow, 21; 43; 59
Data Flow acyclique, 63
data-driven, 60
define, *45*
demand-driven, 60
dictionnaire des symboles, 45
durée de vie des objets, 45
entrées/sorties, 17
environnement, 42; 45; 74
environnement local, 49
environnement temporaire, 75
espace adressable, 19
éta-conversion, 30
état d'un processus, 59
évaluateur, 72; 73
évaluation paresseuse, 43
Fast Fourrier Transform, 19
fichiers compilés, 77
flot de contrôle, 62
fonction anonyme, 27
fonction Eval, 36
fonction récursive, 33; 50
fonction Y, 34
forme normale, 31
formes spéciales, 45
Gabriel, 10
garbage-collecting, 45; 68; 70
heap, 68
horloge, 62
identificateur, 75
indirection, 75
instruction, 18
instruction de répétition, 16
interruption, 16
IOR, Input Output Rate, 62
iswim, 42
lambda, *47*; 74
lambda-abstraction, 26
lambda-expression, 25
langage fonctionnel, 21
let, 76
lexèmes, 72
liaison statique, 42; 73
librairies à liens dynamiques, 77
lisp, 41
liste, 46
liste vide, 46
listes infinies, 43
machine de Turing, 24
mode d'adressage, 19
nombre de paramètre, 70
opérateur de recyclage, 63
paire, 66
parallélisme asynchrone, 20
parallélisme synchrone, 20
paramètre formel, 26
pile, 68; 70
pipeline, 18
point fixe, 34
fonction, 44

quotation, 47
radicaux, 25
récursion de queue, 52
réduction, 25
réduction en ordre normal, 33
règle d'évaluation, 45
représentation externe, 47
ressource partagée, 17; 20
Scheme, 43
sémantique dénotationnelle, 36
sémantique formelle, 42
sémantique opérationnelle, 35
stack, 68; 70
symbole, 37; 73
système mono-DSP, 17
système multi-DSP, 17
table des symboles, 69
tail-recursion, 52
tas, 68
temps de calcul, 20
théorèmes de Church-Rosser, 32
théorie des domaines, 37
théorie des ordinateurs, 58
timer, 16
TMS 320C25, 16
Transe, 10
type latent, 44
valeur initiale, 61
variable libre, 27; 49
variable liée, 27; 49
vecteur, 69; 74
Von Neumann, 58