Université Paris Sud
Master 2004-2005
Compétences Complémentaires en Informatique

Principes des Langages de Programmation

Christine Paulin-Mohring

Préambule

Ce cours introduit les principes de base de la programmation. Il présente quelques notions de *génie logiciel* précisant des méthodes pour produire des programmes fiables, stables et réutilisables. Il introduit également les notions essentielles de la *compilation* c'est-à-dire le *processus* qui transforme une *suite de caractères* représentant un *programme* en une suite *d'instructions machine* qui pourront s'exécuter pour produire le *résultat* du programme. Les concepts seront illustrés sur les langages C et Fortran.

Ce cours s'appuiera sur le cours d'algorithmique en précisant la mise en œuvre dans un langage de programmation des structures de données et algorithmes présentés ainsi que sur le cours d'architecture en ce qui concerne la production de code machine dans la compilation.

Les travaux pratiques associés à ce cours seront l'occasion de se familiariser avec des outils de développement logiciel, en particulier pour faciliter l'édition, la gestion de version, la compilation et le diagnostic d'erreurs.

Quelques informations utiles

La page WEB du cours : `http://www.lri.fr/~paulin/CCI`

Evaluation : L'évaluation comportera les éléments suivants :
- 1 TP noté à rendre le lundi 11/10
- 1 TP noté à rendre le lundi 18/10
- 1 projet à rendre le vendredi 29/10
- 1 examen écrit la semaine du 1er novembre
- 1 épreuve pratique à la fin du semestre

Coordonnées :

Adresse électronique	`Christine.Paulin@lri.fr`
Téléphone	01 69 15 66 35
Bureau	LRI Bat 490, bureau 149.

Merci d'utiliser prioritairement le courrier électronique.

Table des matières

Chapitre 1

Introduction

1.1 Préambule

L'objectif de ce cours est de présenter des concepts et méthodes utiles à l'activité de programmation et plus particulièrement à la maîtrise des langages de programmation.

Programmer, c'est fournir à un ordinateur un ensemble d'instructions et de données permettant la résolution d'un problème donné.

L'activité de programmation est intrinsèquement difficile. Programmer prend beaucoup de temps. Notoirement, la majorité des programmes comporte des erreurs, ne résout que partiellement les problèmes qu'on se pose. Les programmes sont souvent complexes à utiliser ou bien encore difficiles à faire évoluer.

Dans de nombreux cas, il est essentiel d'appliquer de bons principes de programmation.

- Lorsque les programmes sont critiques, c'est-à-dire que les erreurs potentielles peuvent avoir des conséquences humaines ou financières graves (chirurgie assistée par ordinateur, programmes embarqués dans les avions ou les automobiles, commerce électronique). L'explosion au premier lancé d'Ariane 5 était due à une erreur de programmation lié à une conversion non valide d'un flottant en un entier ...
- Lorsque les programmes sont volumineux, ils doivent être structurés en composants largement indépendants. La réutilisation de bibliothèques fiables et efficaces permet de gagner du temps de développement et peut augmenter la qualité du code final.
- Lorsque le programme est développé par plusieurs personnes simultanément ou dans le temps, la documentation du développement, la lisibilité du code sont des facteurs important. Une particularité du code logiciel est d'évoluer de manière très rapide. Ceci est dû d'une part à l'évolution du matériel et des langages de programmation sur lesquels s'appuie le programme, d'autre part à la nécessité d'adapter les fonctionnalités du logiciel aux besoins des utilisateurs.
- Lorsque le programme a des utilisateurs qui ne sont pas les développeurs du programme, il convient d'offrir une interface adaptée au public visé. L'interface prend en charge l'entrée des données que devra traiter le programme ainsi que l'affichage des résultats. C'est à ce niveau que l'on devra prendre en compte l'interaction de l'outil développé avec d'autres programmes proposant leurs propres formats d'entrées/sorties. Le programme doit également gérer l'affichage des erreurs, que ce soit celles de l'utilisateur ou bien les erreurs du programme lui-même.

Quelques chiffres Le groupe américain Standish `www.standishgroup.com` effectue des analyses sur le développement des applications informatiques dans les entreprises américaines. Dans son rapport de 1994, il fournit les chiffres suivants :

- 16 % des projets informatiques sont réalisés dans l'enveloppe budgétaire et les délais prévus et réalisent les fonctionnalités initialement attendues.
- 31 % des projets n'aboutissent pas.
- les 53 % de projets restants sont achevés mais dans des délais deux fois plus longs en moyenne avec un coût presque double et avec seulement 60 % des fonctionnalités souhaitées.

Le coût de la détection et correction d'erreurs est estimé à près de 80% du coût total de développement d'un logiciel. Néanmoins, toutes les erreurs dans les programmes mis sur le marché ne sont pas corrigées, on considère qu'en moyenne 15 % de ces erreurs sont encore présentes dans les produits distribués.

Les sociétés souffrent également de devoir utiliser des programmes anciens qu'il faut maintenir ou moderniser sous peine de devoir les abandonner.

Les critères d'un bon programme Un programme doit donc répondre aux besoins de ses utilisateurs (ce qui nécessite de bien identifier les attentes en terme de fonctionnalité et d'interface); il doit fonctionner correctement (ce qui demande de détecter et corriger les erreurs systématiquement); il doit également être capable d'évoluer dans le temps en fonction des évolutions du matériel et des conditions d'utilisation (ce qui nécessite de faire des choix technologiques pérennes; d'adopter une architecture logicielle évolutive et de documenter largement les options de développement).

1.2 A propos de ce cours

1.2.1 Objectifs

Un bon programmeur est celui qui est capable de maîtriser les étapes d'analyse du problème, qui choisira un bon découpage modulaire du système, qui saura utiliser les outils, langages et technologies adéquats et qui produira in fine un code et une documentation clairs.

Il est essentiel de maîtriser les concepts (modèles d'architectures, algorithmes et structures de données) plutôt qu'un langage particulier, en effet les langages de programmation évoluent dans le temps et peuvent être imposés par l'environnement de développement.

Mais il faut également être capable de transmettre des ordres à un ordinateur en utilisant au mieux ses capacités. Or il y a d'une part ce que l'on veut faire faire à la machine, d'autre part ce que l'on dit à la machine de faire et finalement ce que la machine comprend de ce qu'on lui dit et comment elle le traduit en une suite d'instructions machines. Chaque étape induit une transformation de l'information qui peut avoir des conséquences fâcheuses si elle ne préserve pas le sens des calculs. Cependant, si les ordinateurs ne font pas toujours ce que l'on veut, il font en général ce qu'on leur dit. Maîtriser le langage des ordinateurs est donc un passage obligé du programmeur.

1.2.2 Plan

Dans la suite de ce cours, on s'intéressera tout d'abord aux différentes étapes de la compilation qui permet de passer de la description d'un programme sous forme d'une suite de caractères à une suite d'instructions machine. On détaillera en particulier les outils permettant de décrire un langage. Nous présenterons ensuite les principales caractéristiques du langage C, un langage largement utilisé, très utile pour faire de la programmation proche du système et des processeurs. Nous illustrerons également les concepts de programmation sur le langage Fortran, utilisé principalement dans les programmes de calcul scientifique.

Chapitre 2

Quelque éléments de Génie Logiciel

2.1 Les étapes de la réalisation d'un programme informatique

La suite de ce chapitre présente quelques éléments de Génie Logiciel. Il s'inspire partiellement du début du cours de Génie Logiciel de Marie-Claude Gaudel en licence d'Informatique.

La réalisation d'un programme informatique passe par différentes phases.

2.1.1 Analyse des besoins

On commence en général par l'expression d'un besoin général (par exemple la gestion informatisée des emplois du temps) qu'il faut ensuite analyser afin de préciser les contours du produit à développer et d'éviter le développement d'un logiciel inadapté.
On doit déterminer qui utilisera le programme ? pour quelles opérations ? dans quel environnement matériel et logiciel ?
Cette analyse des besoins aboutit à la définition d'un *cahier des charges* qui décrit ce que doit faire le système (mais qui ne s'intéresse pas dans un premier temps à la manière dont le programme réalisera les opérations). Ce cahier des charges peut se voir également comme un squelette du manuel d'utilisation du système. La précision et la pertinence de cette analyse est un facteur déterminant du succès du développement.
Dans notre exemple de gestion d'emplois du temps, on devra préciser si le programme doit prendre en charge la réservation des salles, s'il est à destination des étudiants, des enseignants, dans quelles conditions il peut être consulté, modifié etc.
Cette analyse des besoins, faite en concertation avec les utilisateurs potentiels (en général non-informaticiens) est menée de manière informelle, en langage naturel.

2.1.2 Spécification globale

Il s'agit d'obtenir une première description du futur projet qui prenne en compte à la fois les besoins mais également des considérations techniques liées au traitement informatique. On précise ce que fera le système mais sans prendre de décisions prématurées quant à la réalisation.
Des notations telles que UML (Unified Modeling Language) peuvent servir de support à cette étape de spécification. Différents types de diagrammes permettent de spécifier et de préciser le systèmes.

- Les *diagrammes de classes* précisent les entités concrètes ou abstraites manipulées par le programme. Dans notre exemple d'emploi du temps on aura certainement à manipuler des *Utilisateurs*, des *Salles*, des *Créneaux horaires*. Une salle pourra être associée à une localisation mais aussi à une capacité d'accueil et un équipement spécifique. Une réservation est une entité qui fait intervenir un utilisateur, une salle et un ou plusieurs créneaux horaires, il peut lui être associé un numéro de réservation. L'écriture systématique de ces diagrammes permet de découvrir certaines ambiguïtés ou imprécisions dans les spécifications qui pourront être éclaircies en interagissant avec les utilisateurs. On peut également à ce niveau mettre en évidence des contraintes d'utilisation du système (pas deux utilisateurs au même moment dans la même salle) qui doivent être pris en compte dans le design de l'application.

- Les *diagrammes de cas d'utilisation* précisent les opérations que peuvent réaliser les utilisateurs d'un système. Par exemple, notre programme de gestion d'emplois du temps pourra être en consultation libre mais les activités de réservation ou annulation peuvent être restreintes à certaines personnes.
- Les *diagrammes de séquences* précisent le déroulement temporel des opérations. Ils permettent de préciser les échanges de messages et les cas d'erreur. Dans le cas où on fait une réservation, on pourra préciser les étapes d'authentification permettant de vérifier que l'utilisateur a le droit de réserver, on pourra ensuite préciser les contraintes qui pourraient amener à rejeter la réservation (données non valides, salle déjà réservée ...) éventuellement émettre des diagnostics (l'utilisateur a déjà une autre réservation au même moment) et enfin, si tout se passe normalement renvoyer un numéro de réservation.

Des langages tels que UML sont dits semi-formels, en effet il s'agit de dessins, dont on comprend le sens général mais qui ne disposent pas d'une interprétation bien définie. Dans certains cas de code particulièrement critique, on peut faire appel à des langages de *spécifications formelles* qui permettent de préciser dans un langage mathématique le comportement attendu des programmes. Cette approche est particulièrement adaptée dans le cas de programmes réalisant des fonctions mathématiques bien identifiées (résolution d'équations, manipulations algébriques ...) ou des propriétés de sécurité sur des systèmes ou des données (les barrières du passage à niveau doivent être fermées lorsque le train est dans une certaine zone, une salle ne doit pas être réservée au même moment par deux utilisateurs différents).

2.1.3 Conception architecturale

C'est l'étape de décomposition du logiciel en composants plus simples. On pourra à cette étape faire le choix de la réutilisation d'outils existants que ce soit des bibliothèques ou bien des environnements spécialisés comme un *Système de Gestion de Bases de Données (SGBD).* On peut aussi choisir de réutiliser des technologies du WEB pour réaliser les interfaces. Certains des choix de conception (par exemple le langage d'implémentation) pourront être imposés par l'environnement dans lequel le programme doit s'insérer. On précisera les interfaces de chacun des composants (l'interface est l'ensemble des données et des opérations fournies par un composant). Ce découpage permet de préparer les tests d'intégration qui permettront de vérifier les interactions entre les composants.

Parmi les bons principes à mettre en œuvre, on retiendra tout particulièrement le fait de séparer les fonctionnalités du système des opérations réalisant l'interaction entre l'utilisateur et le système. Dans le cas des emplois du temps, on aura une opération de réservation élémentaire qui suppose donnés un utilisateur, une salle et un créneau horaire et agit sur le système en effectuant la réservation. Une telle opération peut supposer que le format de ses données est correct et effectuera éventuellement la vérification de disponibilité de la salle. La manière dont ces données sont fournies au système doit être réalisée de manière indépendante : ce peut être un fichier d'entrées, on peut aussi interagir avec une interface WEB qui pourra proposer un ensemble de possibilités ou encore via une interface graphique. Il est essentiel que les présupposés sur l'utilisation de chaque opération d'un composant soient clairement identifiés afin de prévenir une utilisation erronée.

2.1.4 Conception détaillée

La conception détaillée consiste à décrire pour chaque composant la manière dont les fonctions vont être réalisées. On précise alors la représentation des données ainsi que les algorithmes qui vont être utilisés. Les fonctions du module vont être décomposées en fonctions plus simples. Chaque fonction pourra être mise au point en utilisant des jeux de *tests unitaires*.

2.1.5 Programmation

L'étape de programmation consiste à traduire les opérations décrites au moment de la conception en programmes qui peuvent être exécutés par une machine.

Génération automatique de code Cette étape peut parfois être réalisée de manière automatique si la conception a été conduite dans des langages de descriptions formels. C'est le cas de la méthode B de J.-R. Abrial [1, 7] qui est utilisée en particulier pour le développement de logiciels de transports (Meteor) et qui propose de décrire les spécifications et les différentes étapes de la conception sous la forme de *machines* composées d'un état et d'opérations agissant sur cet état dont la description est purement déclarative à l'étape de spécification (on décrit la propriété attendue du résultat de l'opération, pas comment on le calcule) et au fur et à mesure de la conception devient de plus en plus opérationnelle. Le résultat final peut directement être traduit vers du code C. L'utilisation de langages formels permet au système d'engendrer des conditions suffisantes qui garantissent que le programme final répond bien aux exigences de la spécification.

De même des langages évolués tels que Esterel ou Scade permettent de décrire des systèmes réactifs dans des formalismes mettant en évidence les propriétés du programme et fournissant des méthodes de vérification et de compilation vers du code embarqué.

Conception et programmation Si certains langages de spécification évolués peuvent être traduits automatiquement en code exécutable; il est inversement assez courant de réaliser l'étape de conception directement via la programmation. En effet les langages de programmation modernes mettent l'accent sur la structuration des développements (découpage en modules, objets, qui bénéficient d'une compilation séparée), sur l'explicitation d'interfaces et sur des instructions de haut niveau. Les interfaces traduisent les choix de la conception architecturale. Le détail des opérations à implanter dépendra directement des bibliothèques que l'on choisira de réutiliser. Cette approche plutôt pragmatique a le défaut de rendre le développement dépendant du langage de programmation choisi. Par contre, la précision des langages de programmation peut être un avantage, la possibilité de manipuler mécaniquement les programmes permet d'effectuer automatiquement certaines vérifications. Il serait également peu judicieux de garder deux entités séparées : la conception détaillée et le programme dont l'un serait une paraphrase de l'autre. En effet garantir la cohérence des deux versions au fur et à mesure des évolutions peut devenir très complexe.

L'étape de conception, même si elle a pour résultat la production de code exécutable, reste un élément essentiel du développement du programme. Les choix qui sont faits doivent donc être clairement explicités, tout comme on explique la méthode et le résultat attendu avant de se lancer dans un gros calcul. Le code doit être commenté de manière pertinente pour assurer sa lisibilité.

2.1.6 Vérification

La vérification consiste à s'assurer de la correction des programmes développés par rapport aux spécifications détaillées. Cette vérification se fera d'abord au niveau de chaque opération

dans chaque composant et ensuite après l'étape d'intégration des composants qui permet d'obtenir une programme exécutable.
Il y a différentes techniques de vérification. Certaines travaillent uniquement sur le source du programme, on parle alors de *vérification statique*. D'autres nécessitent l'exécution du code produit, on parle alors de *vérification dynamique*

Vérification statique

Il peut s'agir d'inspecter le source du programme ou plus généralement les documents de spécification et de conception (ce que font les organismes de certification, ou les correcteurs de projets ...).

On peut aussi effectuer mécaniquement certaines vérifications sur le code. Le contrôle de types dans les compilateurs des langages évolués est un exemple de vérification statique. Malheureusement, on ne peut pas garantir automatiquement l'absence complète d'erreurs dans un programme si celui-ci s'exprime dans un langage assez général. Par exemple, savoir si un programme va s'arrêter sur une entrée donnée ou si le programme peut provoquer une division par zéro ou un accès en dehors des bornes d'un tableau sur certaines entrées sont des propriétés dites indécidables. C'est-à-dire qu'il n'existe pas de programme de diagnostic qui prenne en entrée la description d'un programme et qui réponde à coup sûr :

- Oui l'exécution de ce programme est sûre sur n'importe quelle entrée (pas de division par zéro et d'accès en dehors des bornes du tableau).
- Non il existe des entrées pour lequel le programme provoque des erreurs.

Le mieux que l'on sache faire en matière de vérification automatique, c'est émettre des diagnostics partiels : oui tout marche bien, non voici un exemple où cela ne marche pas mais aussi peut-être que cela marche, peut-être que cela ne marche pas ...

Évidemment, pour un programme donné, on peut exprimer l'absence d'erreurs sous la forme d'une formule mathématique qu'il est en principe possible de prouver (mais pas toujours de manière automatique).

Tests

La méthode de vérification la plus répandue consiste à effectuer des tests de manière dynamique en exécutant le code développé. Les tests doivent être prévus dès la phase de spécification du système. Il est recommandé d'automatiser autant que possible les jeux de tests de manière à pouvoir les exécuter simplement de manière systématique. Lorsque l'on conçoit des jeux de test, on doit en particulier prévoir :

- des cas correspondants aux conditions limites d'utilisation du système, en particulier des cas d'erreurs.
- des cas correspondant à des parcours systématiques du code du programme (par exemple s'assurer que l'on explore systématiquement les deux branches d'une conditionnelle).

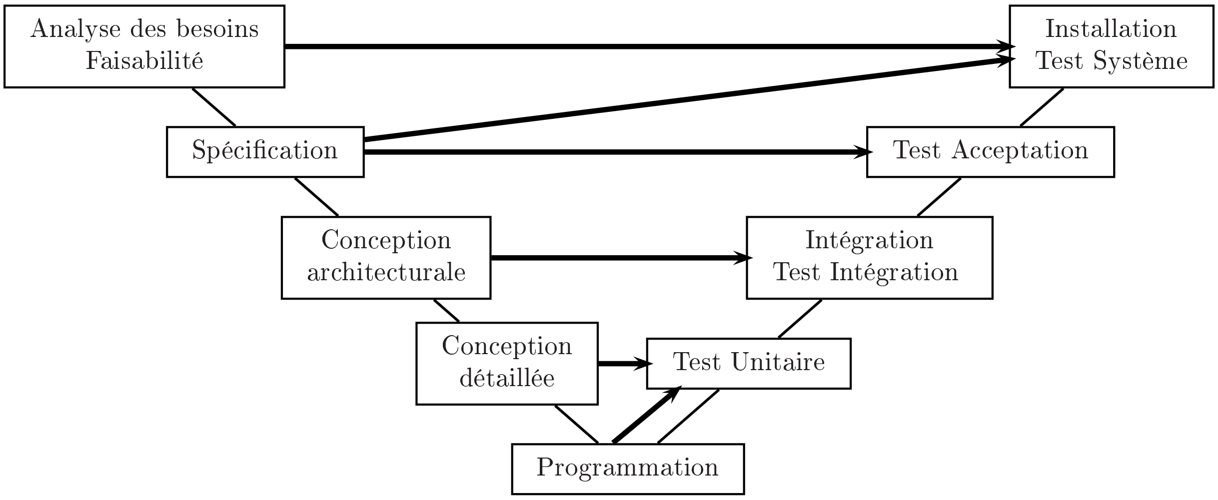
2.1.7 Validation

La validation consiste à s'assurer que le système développé répond aux besoins de l'utilisateur et s'insère dans son environnement matériel d'utilisation. C'est un processus qui en général met en œuvre des composants qui ne sont pas informatiques s'il faut par exemple mettre le logiciel en place dans un avion ou bien au sein d'une expérience physique ou encore face à des utilisateurs novices...

2.2 Modèles de développement

Le modèle de développement décrit l'organisation des étapes du développement de logiciel. En effet certaines tâches ne peuvent être démarrées avant que d'autres soient terminées, il faut également détecter et corriger les erreurs de spécification ou de conception le plus tôt possible.

Le modèle en V Ce modèle classique est largement utilisé. Il se parcourt de gauche à droite, les phases de spécification et de conception sont à gauche, les phases de vérification et de validation sont à droite. Les phases se correspondent à chaque niveau.



2.3 Documentation

La documentation d'un logiciel est une activité importante qui intervient à toutes les étapes du développement. Le système doit comporter un *Manuel d'utilisation* à destination des utilisateurs et un *Manuel de référence* qui servira à la maintenance du système. On peut également avoir à établir des rapports intermédiaires d'avancement afin d'aider à la gestion en terme de ressources et à la planification du projet.

2.3.1 Manuel d'utilisation

Un manuel d'utilisation pourra contenir les éléments suivants :

- Identification du logiciel : nom, numéro de version, auteurs, hotline ...
- Introduction : description générale, possibilités et limites, terminologie, plan du manuel, référence
- Environnement du logiciel : environnement matériel minimal conseillé, environnement logiciel, systèmes d'exploitation
- Installation/Démarrage du logiciel : manipulations du logiciel, connexion de périphériques, chargement de fichiers, paramétrage ...
- Guide de l'utilisateur : les principales fonctions du système illustrées sur des exemples simples
- Liste des commandes : messages, cas d'erreurs. Éventuellement format des fichiers.

2.4 Tests

2.5 Quelques règles pour l'écriture des programmes

Ces règles proposant des conventions pour l'écriture de programmes lisibles et évolutifs sont largement inspirée du livre [2].

- Choisir des noms de variable qui reflètent leur rôle. Utiliser des noms cours pour les variables locales, ou les indices de boucle. Utiliser des plus longs pour les variables externes.

Chapitre 3

Introduction à la compilation

Ce chapitre présente les différentes phases de la compilation et introduit les éléments permettant de décrire un langage de programmation.

3.1 Introduction

Une machine permet d'exécuter une suite d'instructions élémentaires (opérations arithmétiques, accès mémoire, branchements conditionnels) sur des données qui sont des suites de 0 et de 1 qui seront interprétées comme des entiers, des flottants, des caractères ou encore des adresses mémoires.

Heureusement, la description d'un calcul à effectuer ou d'une donnée peut se faire dans des langages de haut niveau.

On écrit le caractère 'A' qui sera représenté suivant le code ASCII par l'entier 65 qui est codé de manière binaire par 01000001, ce qui se lit plus facilement en utilisant la notation hexadécimale :41.

On communique donc avec la machine dans un langage qui peut être écrit, lu et compris par des humains. Un programme particulier (appelé compilateur ou interpréteur) se charge d'engendrer une suite d'instructions machines qui pourra être exécutée afin de fournir un résultat.

On parle d'*interpréteur* dans le cas d'un exécutable qui prend en entrée le texte du programme à exécuter décrit dans un langage de haut niveau et l'entrée de ce programme et calcule le résultat attendu.

Un compilateur est également un exécutable qui prend en entrée le texte du programme à exécuter et engendre une suite d'instructions machine qui pourra elle-même être exécutée sur différentes entrées.

Écrire un interpréteur est en général plus facile à faire qu'écrire un compilateur. Les langages dits de *scripts* (shell, python...) sont interprétés. Un compilateur produira en général un code plus efficace, par contre le code produit doit être différent suivant les architectures cibles visées. Certains langages adoptent des stratégies mixtes. Ainsi la philosophie de Java est de compiler le source vers un format intermédiaire (le byte-code) qui est largement portable et qui est interprété par un programme appelé *machine virtuelle Java*.

3.2 Les différentes phases de la compilation

Nous allons maintenant nous intéresser aux différentes étapes de la compilation d'un programme.

Nous prendrons comme exemple, le programme suivant écrit dans le langage C.

```
/* Bibliothèque standard */
```

```
#include <stdio.h>

/* Programme principal */
main()
{
printf("bonjour maître \n");
}
```

La première grande phase est l'*analyse*. Elle consiste à *comprendre* la nature du calcul qui est demandé et à détecter d'éventuelles erreurs. Ensuite vient la phase de *génération de code* machine. Il s'agit de déterminer une suite d'instructions machines qui effectuera le calcul décrit par le programme d'entrée. On cherchera dans cette phase à produire un code évidemment correct mais également efficace soit en terme de taille de code (important dans les systèmes embarqués) soit en terme de ressources mémoire ou de temps de calcul. Finalement, il y a une phase de production d'un *code exécutable*. Le *chargement* est l'opération système qui fait passer le code exécutable dans la mémoire de l'ordinateur. Les adresses dans le code exécutable qui pour des raisons de portabilité sont stockées sur disque de manière relatives sont transformées en des adresses absolues. Nous détaillons maintenant ces grandes étapes.

3.2.1 L'analyse

La phase d'analyse peut démarrer par une phase dite de pre-processing qui effectue des remplacements élémentaires dans le texte du programme. Par exemple la directive `#include <stdio.h>` indique au pre-processeur d'inclure à cet endroit le contenu du fichier `stdio.h` qui contient en particulier la déclarations de la fonction externe `printf` utilisée dans le corps du programme. On utilise également le pre-processeur pour définir des constantes ou bien engendrer à partir du même fichier plusieurs codes analogues à utiliser dans des contextes différents.

La phase d'analyse se décompose en une phase d'*analyse lexicale* qui consiste à repérer les différents *mots* du langage puis une phase d'*analyse syntaxique* qui va identifier les *phrases* et finalement une phase d'*analyse sémantique* qui cherche à identifier le *sens* du programme (quelle opération cherche-t-on à réaliser). Chaque phase a pour rôle de rejeter des programmes jugés incorrects et de traiter son entrée afin de faciliter le travail à l'étape suivante.

Lors de l'analyse lexicale, on va repérer les *mot-clés* du langage qui sont essentiels à la reconnaissance des phrases. On va également reconnaître les identificateurs et les constantes (numériques ou chaînes de caractères) qui seront éventuellement prétraitées. L'analyse lexicale va se débarrasser des éléments qui ne servent pas au calcul comme les commentaires et autres éléments de mise en page.

Lors de l'analyse syntaxique, on se chargera de déterminer la structure du programme, en particulier on lèvera les ambiguïtés dans l'écriture des expressions par exemple en utilisant les règles de précédence du langage (l'expression $3+z-2$ est équivalente à $(3+z)-2$). On cherchera une représentation unique pour des constructions qui s'écrivent différemment mais qui ont le même sens. On représente en général le programme sous une forme arborescente qui met en évidence sa structure et qui est plus appropriée à l'expression de son sens que la forme linéaire du texte.

Lors de l'analyse sémantique, on détermine le sens du programme. Une étape importante est de gérer les noms de variables ou de fonctions apparaissant dans le programme : quelle taille mémoire sera nécessaire pour stocker une variable, à quelle définition fait référence un appel de fonction (plusieurs calculs différents peuvent porter le même nom et c'est le contexte d'utilisation qui peut permettre de résoudre l'ambiguïté).

L'analyse est un procédé déterministe que la machine réalise pour associer un calcul au texte d'un programme. Cette analyse dépend évidemment du langage de programmation source même si un certain nombre de mécanismes sont communs à différents langages. Maîtriser cette chaîne d'analyse est essentiel pour décrire à la machine le calcul que l'on souhaite faire.

S'il est rare d'avoir à développer un compilateur complet, il est de nombreuses situations dans lesquelles on a besoin d'outils d'analyse afin de traiter l'entrée d'un programme ou bien d'effectuer des transformations de formats de données. Il est donc utile de connaître les principales techniques et outils qui permettent d'effectuer ces analyses simplement. Nous reviendrons sur les méthodes et outils pour réaliser ces opérations dans la suite de ce cours.

3.2.2 Génération de code

La phase de génération de code passe par des étapes de production de codes intermédiaires. Ces codes permettent de traiter les aspects avancés du langage comme la représentation des données structurées ou la décomposition des expressions numériques en opérations binaires. Ces langages intermédiaires sont indépendants de la machine cible. Une étape importante de la génération de code est la gestion de la mémoire. Il faut organiser les données dans la mémoire de manière rationnelle. Certains des choix dépendent du programme et du langage. Lorsque l'on est sûr de manière statique (en analysant le texte du programme) qu'une certaine variable ne sera plus utilisée, il est possible de récupérer l'espace mémoire correspondant pour stocker d'autres données. Une partie de la mémoire est ainsi généralement gérée sous la forme d'une *pile* : lorsque l'on a besoin d'espace mémoire, on le réserve au sommet de la pile, lorsque l'objet ainsi stocké n'est plus utile, on le *dépile* et l'espace ainsi libéré peut être réutilisé pour un autre objet.

On fait finalement les étapes qui dépendent de l'architecture de la machine cible.

3.2.3 Vers un processus exécutable

La compilation peut souvent se faire de manière modulaire, fichier par fichier. Cependant, toutes les fonctions utilisées dans un fichier ne sont en général pas définies dans ce fichier. C'est le cas des bibliothèques externes en particulier. Au moment d'exécuter un programme, il faudra néanmoins que tout soit défini. C'est le rôle de l'édition de liens qui va créer un fichier exécutable en rassemblant les codes compilés des différents modules et celui des bibliothèques appelées. L'édition de liens va transformer les références symboliques aux fonctions externes en des références relatives qui dépendent de l'organisation des modules. L'édition de liens peut-être statique ou dynamique. Dans le cas statique, les codes correspondant aux fonctions des bibliothèques vont être recopiés dans l'exécutable du programme ce qui fait grossir la taille de l'exécutable. Dans le cas dynamique la liaison aura lieu au moment du chargement du programme ou encore à l'exécution ce qui pourra par contre ralentir ces opérations.

3.2.4 Les commandes de compilation en Fortran et C

Un programme C ou Fortran est écrit dans un fichier qui doit être ensuite compilé à l'aide d'un programme spécifique appelé compilateur. Suivant les options passées au compilateur, cette commande produira différents fichiers.

Le cas des programmes C compilés avec `gcc`

Dans le cas d'un programme C, le fichier aura une extension `.c`, par exemple `essai.c`. On décrit ici le fonctionnement du compilateur `gcc` (Gnu C Compiler).

La commande shell :

```
gcc essai.c
```

va d'abord effectuer une phase de pre-processing qui élimine certaines directives et définitions de constantes puis va compiler le fichier. S'il n'y a pas d'erreurs, la compilation produit d'abord un code assembleur (par défaut `essai.s`), puis un fichier binaire qui peut être reloger (par défaut `essai.o`). Finalement, le compilateur effectue l'édition de lien et produit un exécutable (par défaut `a.out`).

Pour exécuter le programme, il faut taper la commande :

```
./a.out
```

en passant éventuellement sur la ligne de commande les arguments attendus par le programme.

La compilation efface au fur et à mesure les fichiers intermédiaires. Des options de compilation permettent de n'effectuer qu'une partie de la compilation :

`gcc -S`	effectue les étapes de compilation jusqu'à produire le code assembleur
`gcc -c`	effectue les étapes de compilation jusqu'à produire le code binaire relogeable, sert à la compilation séparée

L'option `-o` *mon_programme* permet de renommer le fichier de sortie produit par `gcc`.

D'autres options contrôlent les optimisations ou encore le sous-ensemble de C accepté en entrée, la nature des vérifications statiques qui sont réalisée ou bien encore la production d'un exécutable pouvant être utilisé avec des outils de debug. Se reporter au manuel (`man gcc`) pour le détail des options.

Le cas des programmes Fortran

Il y a plusieurs langages Fortran et donc plusieurs compilateurs. Dans le monde des logiciels libres, `g77` est le compilateur GNU pour Fortran 77. Il partage l'architecture de `gcc`. Les options sont identiques, sauf que l'on part d'un fichier fortran dont le suffixe est `.f` ou `.for` ou encore `.fpp` lorsque les fichiers doivent être préalablement traités par le pré-processeur `cpp`.

Le compilateur `F` est produit par la compagnie *Fortran Company / NAGWare*, il permet de compiler des fichiers écrits en Fortran 90 ou Fortran 95. Le suffixe de ces programmes est `.f90` ou `.f95` (ou bien `.F90` ou `.F95` dans le cas où un pre-processing est requis). Les options suivantes peuvent être utilisées :

`F -S`	effectue les étapes de compilation jusqu'à produire un code C (à compiler encore avec F)
`F -c`	effectue les étapes de compilation jusqu'à produire le code binaire relogeable, sert à la compilation séparée

Comme pour `gcc`, l'option `-o` *mon_programme* permet de renommer le fichier de sortie produit par `F`.

3.3 Notations syntaxiques

La présentation d'un langage de programmation commence par la description de sa syntaxe, c'est-à-dire des règles qui décrivent les programmes correctement formés qui seront acceptés par le compilateur.

3.3.1 Terminologie

Dans le domaine des descriptions de langage de programmation, on utilise un vocabulaire largement inspiré des constructions grammaticales du langage naturel.

Ainsi, un *Alphabet* représente n'importe quel ensemble fini. Ce peut-être les caractères qui peuvent être entrés au clavier : non seulement les caractères alphabétique, ou numériques mais également les différents symboles typographiques, les espaces, les retours à la ligne, ou d'autres caractères non affichables. L'alphabet peut également être un ensemble *d'entités lexicales* (identificateurs, mots-clés ...) qui sert d'entrée à l'analyseur syntaxique.

L'élément d'un alphabet s'appelle une *lettre*. Un *mot* est une suite finie de lettres. On utilise également le mot vide qui ne contient aucun caractère. Un *langage* est un ensemble de mots.

3.3.2 La notation BNF

La notation BNF (Backus-Naur Form) a été mise au point dans les années 60 lors de la définition du langage Algol. Elle est largement utilisée (avec des variantes) pour décrire de manière précise la syntaxe des langages de programmation.

Un programme est une suite de caractères. Certaines suites de caractères ont un rôle très spécifique (symboles particuliers, mot clés) mais d'autres ont un rôle plus général et peuvent être organisés en différentes catégories : les constantes entières, flottantes ou de chaîne, les commentaires, les identificateurs, les expressions, les instructions, les types, déclarations ...

Dans la notation BNF, on donne un nom à chacune des catégories intéressantes qui ne sont pas des suites de caractères simples mais qui représentent des ensembles de suites de caractères jouant un rôle analogue.

Une catégorie est représentée par un nom (choisi de manière pertinente par l'utilisateur) encadré par les signes < et >. On aura par exemple les catégories <*constante-entière*>, <*identificateur*> ou encore les catégories <*expression*> et <*instruction*>.

Chaque catégorie définie se verra associée une ou plusieurs définitions qui explicitent comment on peut construire un objet dans cette catégorie à partir de certains caractères spécifiques et d'objets d'autres (ou de la même) catégories. La catégorie définie est séparée de sa définition par le signe spécial `::=`.

Une *définition* s'écrit :

`<catégorie> ::= corps de la définition`

Ainsi pour décrire la catégorie <*chiffre*> correspondant à un chiffre de 0 à 9, on peut utiliser les dix définitions suivantes :

<*chiffre*> `::=` 0	<*chiffre*> `::=` 1
<*chiffre*> `::=` 2	<*chiffre*> `::=` 3
<*chiffre*> `::=` 4	<*chiffre*> `::=` 5
<*chiffre*> `::=` 6	<*chiffre*> `::=` 7
<*chiffre*> `::=` 8	<*chiffre*> `::=` 9

On remarque que la partie droite de la définition est formée d'un simple caractère.

Une notation moins verbeuse consiste à regrouper plusieurs corps des définitions de la même catégorie en les séparant par le signe spécial |. On obtient alors pour les chiffres :

<*chiffre*> `::=` 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Une constante entière est une suite finie (non vide) de chiffres, on pourra l'écrire à l'aide de deux définitions :

<*constante-entière*> `::=` <*chiffre*>
<*constante-entière*> `::=` <*chiffre*><*constante-entière*>

On remarque que la première définition utilise dans la partie droite une catégorie déjà définie. La seconde définition utilise également la catégorie définie dans le corps de la définition. Il s'agit donc d'une définition récursive.

À partir des définitions, on construit une relation de dérivation sur des suites finies de caractères et de catégories. Si on a une telle suite $c_0 \ldots c_n$ et si c_i est une catégorie pour laquelle on a une définition c_i`::=`$d_1 \ldots d_k$ alors on a une dérivation de la suite $c_0 \ldots c_n$ vers la suite $c_0 \ldots c_{i-1} d_1 \ldots d_k c_{i+1} \ldots c_n$. On écrira

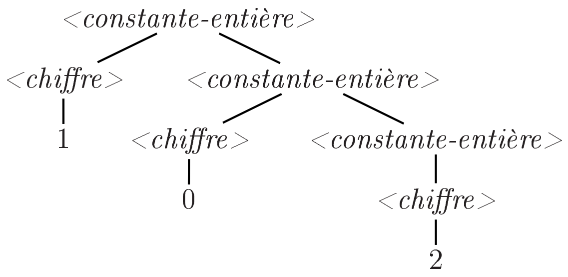
$$c_0 \ldots c_n \to c_0 \ldots c_{i-1} d_1 \ldots d_k c_{i+1} \ldots c_n$$

On peut ainsi effectuer des dérivations à partir d'une catégorie *<cat>* jusqu'à ne plus avoir que des caractères.

Dans le cas de la catégorie *<chiffre>*, les dérivations n'ont qu'une étape. Dans le cas de la catégorie *<constante-entière>*, les dérivations peuvent avoir un nombre arbitraire d'étapes plus grand que 2 :

<constante-entière> → *<chiffre>* → 0
<constante-entière> → *<chiffre><constante-entière>* → 1*<constante-entière>*
→ 1*<chiffre>* → 10
<constante-entière> → *<chiffre><constante-entière>* → 1*<constante-entière>*
→ 1*<chiffre><constante-entière>* → 10*<constante-entière>*
→ 10*<chiffre><constante-entière>* → 102

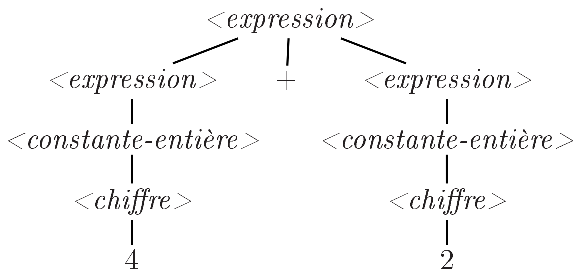
On peut associer à une dérivation, un arbre donc chaque nœud interne a pour étiquette un nom de catégorie et correspond à une définition. Les sous-arbres sont étiquetés par des caractères ou des catégories. Ainsi, la dérivation de `102` à partir de *<constante-entière>* se représente de manière arborescente par :



Le corps d'une définition peut contenir à la fois des catégories et des caractères. Ainsi, la catégorie *<expression>* des expressions arithmétiques pourra contenir les définitions suivantes :

<expression> `::=` *<constante-entière>*
<expression> `::=` *<expression>* `+` *<expression>*

La suite de caractères `4+2` correspond à l'arbre de dérivation syntaxique :



Certaines définitions vont également contenir des mots-clés. Par exemple, la syntaxe de l'instruction d'itération `while` en C se décrit par la règle :

<instruction> `::=` `while (` *<expression>* `)` *<instruction>*

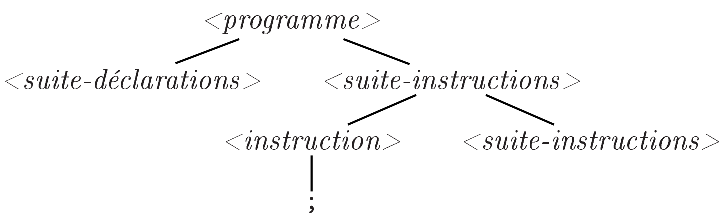
Il est possible que le corps de la définition d'une catégorie soit une suite vide. Ainsi si on veut définir un programme comme une suite de déclarations suivie d'une suite d'instructions, et si le cas où il n'y a pas de déclarations ou pas d'instructions doit être considéré, on pourra écrire la descriptions suivante :

<suite-instructions> `::=` *<instruction><suite-instructions>*
<suite-instructions> `::=`
<suite-déclarations> `::=` *<déclaration><suite-déclarations>*
<suite-déclarations> `::=`
<programme> `::=`*<suite-déclarations>* *<suite-instructions>*

Dans l'arbre de dérivation syntaxique, cela correspond à un nœud sans fils. Dans le langage C, le point virgule est accepté comme une instruction correcte :

<instruction> `::=` `;`

et donc un fichier contenant un unique point-virgule est correct et correspond à la dérivation :



Notations étendues

Il est possible d'utiliser des notations étendues dans les corps de définitions. Ainsi, une partie de la définition peut-être encadrée de crochets carrés [et] : cela signifie que la partie entre crochet est optionnelle. On peut ainsi représenter deux définitions, l'une avec la partie entre crochets et l'autre sans. Ainsi, une instruction du langage *C* peut être formée d'une expression (optionnelle) suivie d'un point-virgule. On peut donc ajouter à notre définition précédente, la définition

<instruction> `::=` *<expression>*`;`

ou bien plus simplement écrire une seule définition :

<instruction> `::=` [*<expression>*]`;`

Dans cette définition, il faut faire la distinction entre le caractère `;` qui appartient au langage que l'on cherche à décrire et les caractères `::=`, [,] ou encore | qui sont des *méta-notations* introduites dans la convention BNF. Comme il peut arriver que ces symboles appartiennent également au langage que l'on cherche à décrire, il convient de les distinguer. On peut le faire en utilisant des fontes différentes, ou bien des conventions comme ajouter des guillemets autour des caractères du langage à décrire utilisés de manière isolée dans le corps d'une définition. Ainsi on pourra écrire :

<instruction> `::=` *<expression>*`";"`
<expression> `::=` *<expression>* `"+"` *<expression>*

Il arrive souvent que l'on doive décrire des catégories formées par répétition d'une autre catégorie. C'est le cas des catégories des constantes entières, des instructions ...Plutôt que d'utiliser deux règles dont une récursive, on peut utiliser des accolades. Si un corps de définition contient une partie entre accolades, cela signifie que cette partie peut être répétée un nombre fini de fois (éventuellement pas du tout). Ainsi on peut définir plus simplement les constantes entières par :

<*constante-entière*> `::=` <*chiffre*> {<*chiffre*>}

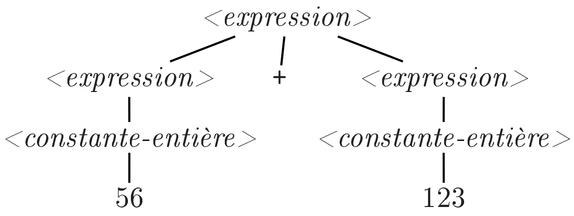
Une suite d'expressions séparées par des virgules pourra se décrire simplement par :

<*suite-expressions*> `::=` <*expression*> {`","` <*expression*>}

3.3.3 Analyse lexicale et expressions régulières

Les langages peuvent en principe être décrits par des règles de syntaxe au format BNF, dans lesquelles on lit des caractères d'entrée. Cependant l'exemple des constantes-entières montre que la notation devient souvent très lourde. D'autant plus que dans nos exemples précédents, nous n'avons pas parlé des caractères d'espacement ou des commentaires. A priori ces caractères peuvent apparaître par exemple autour du symbole `+` dans une expression arithmétique mais évidemment pas au milieu d'une constante entière.

La solution à ce problème consiste à effectuer un prétraitement sur les caractères d'entrées qui va commencer par reconnaître des *unités lexicales*. Ces unités lexicales seront par exemple les constantes entières, les identificateurs, les mots-clés, les commentaires, les séparateurs. . .Certaines de ces unités lexicales (séparateurs, commentaires) vont être ignorées. Les autres vont être considérées comme des caractères généralisés dans la notation BNF. Certaines unités lexicales peuvent correspondre à plusieurs suites de caractères, on les note comme des catégories et on leur associe directement une valeur correspondant à la suite de caractères correspondante. Ainsi si les constantes entières sont considérées comme des unités lexicales, lors de l'analyse de l'expression `56+123`, on associera directement `56` et `123` à <*constante-entière*> sans passer par la décomposition en chiffres. On aura alors l'arbre de dérivation :



Reconnaissance des unités lexicales

L'analyse lexicale transforme la suite de caractères d'entrées en suite d'unités lexicales. Chaque unité lexicale correspond à un langage, c'est-à-dire un ensemble de mots. Cependant, ce langage doit être assez simple pour que la reconnaissance de l'unité lexicale puisse se faire rapidement. Ainsi, lorsqu'on lit un livre, il est aisé grâce aux espaces et aux signes de ponctuation de repérer les mots. Par contre reconnaître la structure de la phrase demande une compréhension globale plus complexe.

Dans les langages de programmation, les constantes entières ou les chaînes de caractères sont des unités lexicales mais pas les expressions ou les instructions. En effet, les unités lexicales vont correspondre à des langages dits *réguliers* dont la reconnaissance peut s'effectuer à l'aide d'automates finis déterministes très efficace.

Notations pour les expressions régulières

On utilise souvent en informatique des *expressions régulières*. Une expression régulière est une notation permettant de représenter un langage reconnaissable par un automate fini (Théorème de Kleene). Dans ce cours, nous nous contentons d'introduire les notations et de définir le langage associé.

Les constructions essentielles des expressions régulières sont la concaténation (mise bout à bout de deux suites de caractères) et l'étoile. À partir d'un expression régulière r, on peut construire une expression régulière r^*, l'étoile de r. Le langage associé à r^* contient le mot vide mais aussi tous les mots obtenus par concaténation d'un mot appartenant au langage associé à r et d'un mot appartenant au langage associé à r^*; c'est-à-dire les mots obtenus en concaténant un nombre fini quelconque de fois (éventuellement 0) des mots correspondant à l'expression régulière r.

Expression régulière : r	Description	Langage associé : $L(r)$
c	Caractère	{c}
$r_1 \| r_2$	Union	$L(r_1) \cup L(r_2)$
$r_1 r_2$	Concaténation	$\{m_1 m_2 \| m_i \in L(r_i)\}$
r^*	Étoile	$\bigcup_{n \in \mathbb{N}} \{m_1 m_2 \dots m_n \| m_i \in L(r)\}$
ϵ	Mot vide	$\{\epsilon\}$
$\emptyset$	Ensemble vide	$\emptyset$

Notations dérivées On utilise très souvent des notations abrégées. L'expression régulière r^+ est une abréviation pour rr^*; le langage correspondant contient les mots formés par concaténation d'un nombre fini non nul de mots appartenant au langage associé à r. De même l'expression régulière $r?$ est une abréviation pour $r|\epsilon$; le langage correspondant contient le mot vide et les mots appartenant au langage associé à r.

Exercice : constantes flottantes Dans le manuel de référence du langage C, il est dit qu'une constante flottante se compose d'une partie entière, d'un point décimal, d'une partie fractionnaire, du caractère `e` ou `E`, d'un exposant entier éventuellement signé et d'un suffixe de type éventuel `f` ou `F` ou `l` ou `L`. Chacune des parties entières ou fractionnaire se compose d'une séquence de chiffres. On peut omettre la partie entière ou la partie fractionnaire (mais pas les deux). On peut aussi omettre le point décimal ou le `e` suivi de l'exposant (mais pas les deux).
Proposer une expression régulière qui corresponde aux constantes flottantes.

Analyseur lexicaux Les analyseurs lexicaux (par exemple `lex`) ou encore certains langages de script permettent de décrire des programmes en associant des actions à des expressions régulières. Le programme lit son entrée et lorsqu'il reconnait une entrée appartenant au langage décrit par l'expression régulière, il déclenche l'action associée en utilisant éventuellement la chaîne reconnue.

Il y a souvent plusieurs manières de découper un texte en unités lexicales. Par exemple `>=` peut être compris comme le symbole $\geq$ ou bien décomposé en deux unités lexicales $>$ et $=$. La convention veut que l'on cherche toujours à reconnaître l'unité lexicale la plus longue.

Exemple de langage non-associé à une expression régulière Les expressions régulières permettent de capturer une classe intéressante de langages, par contre il existe des langages qui ne sont pas réguliers. L'exemple le plus typique de tels langages est celui des expressions parenthésées.

Le langage des expressions que nous avons définis précédemment par les deux règles

<*expression*> `::=` <*constante-entière*>
<*expression*> `::=` <*expression*> `"+"` <*expression*>

Peut se décrire à l'aide d'expressions régulières.

chiffre ≡ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
constante-entière ≡ **chiffre**$^+$
expression ≡ **constante-entière** (+ **constante-entière**)*

Si maintenant, on veut pouvoir parenthéser les expressions, on peut introduire dans la notation BNF une définition :

<expression> `::=` `"("` *<expression>* `")"`

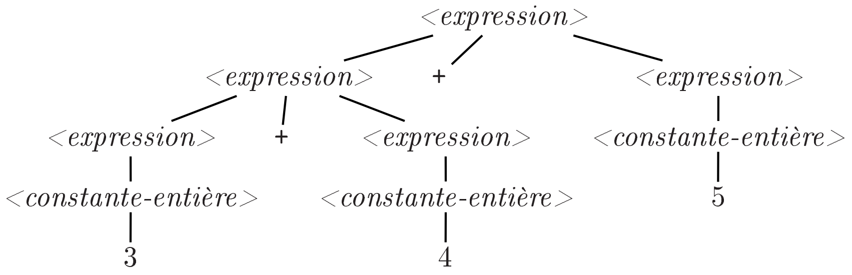
Mais il n'est plus possible de trouver une expression régulière pour décrire le langage. Les expressions régulières ne permettent pas de s'assurer que l'on referme exactement le nombre de parenthèses ouvertes.

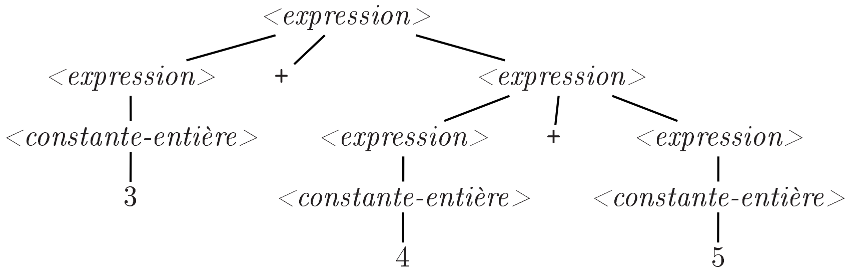
3.3.4 Grammaires

La notation BNF simple (sans option ou répétition) introduite précédemment correspond à la notion de grammaire (hors contexte).

Le vocabulaire des grammaires est un peu différent de celui des grammaires BNF. La grammaire est construit sur deux ensembles finis, l'ensemble T des terminaux qui correspond aux lettres qui forment les mots que l'on veut reconnaître et l'ensemble des non-terminaux qui correspond aux catégories. Les définitions sont appelées règles de production. On identifie un non-terminal particulier appelé symbole initial. La notion de dérivation, ou bien d'arbre de dérivation syntaxique est la même que dans le cas de la notation BNF.

Ambiguïté Beaucoup de grammaires intuitives sont en fait ambiguë, c'est-à-dire qu'une même suite de caractères peut correspondre à plusieurs arbres de dérivation syntaxique. Ainsi l'expression $4+3+5$ correspond aux deux arbres suivants :





Le changement d'interprétation ne change pas le résultat mathématique du calcul. Cependant les problèmes de dépassement de bornes ne seront pas les mêmes dans les deux interprétations. De plus si on met l'opération `-` au lieu de `+` alors les deux arbres ne donnent pas du tout le même résultat. Il est donc essentiel de comprendre ce que fait le compilateur.

Précédences, associativité La notion de *précédence* entre symboles permet de lever les ambiguïtés. Supposons que l'on cherche à analyser une expression de la forme :

$$<\textit{expression}>\ \mathtt{op}_1\ <\textit{expression}>\ \mathtt{op}_2\ <\textit{expression}>$$

et que l'on a les définitions :

$<\textit{expression}>\ ::=\ <\textit{expression}>\ \ \mathtt{op}_i\ \ <\textit{expression}>$

Il y a deux arbres de dérivation qui correspondent à deux parenthésage possibles de l'expression.

(1) (<*expression*> $\texttt{op}_1$ <*expression*>) $\texttt{op}_2$ <*expression*>
(2) <*expression*> $\texttt{op}_1$ (<*expression*> $\texttt{op}_2$ <*expression*>)

Si $\texttt{op}_1$ et $\texttt{op}_2$ sont des opérateurs différents, on peut leur donner des précédences différentes. Les précédences sont juste des entiers. L'idée est de mettre des parenthèses autour de l'opérateur de précédence la plus élevée. Ainsi si la précédence de $\texttt{op}_1$ est strictement plus grande que celle de $\texttt{op}_2$ alors l'interprétation de l'expression correspondra à la version (1). Si c'est au contraire l'opérateur $\texttt{op}_2$ qui a une précédence strictement plus grande que $\texttt{op}_1$ alors on utilisera la seconde interprétation. Par exemple la précédence des signes multiplicatifs $*$ et / est strictement plus grande que celles des opérateurs additifs + et $-$. Si les deux opérateurs ont même précédence, ce sont les règles d'associativité qui déterminent l'interprétation. Une associativité gauche donne la version (1) et une associativité droite donne la version (2).

Enlever les ambiguïtés On peut modifier la grammaire pour respecter les règles de précédence sans ambiguïté. En prenant l'exemple des expressions, on suppose que l'on a classer les opérateurs en 2 classes $\texttt{op}_1$ et $\texttt{op}_2$. On suppose que les opérateurs de $\texttt{op}_2$ ont une précédence plus forte que ceux de $\texttt{op}_1$ et que de plus les opérateurs de $\texttt{op}_1$ associent à gauche et que ceux de $\texttt{op}_2$ associent à droite.

On s'intéresse au langage associé à la description BNF (ambiguë) suivante :

<*expression*> `::=` <*constante-entière*> | `"("` <*expression*> `")"`
| <*expression*> $\texttt{op}_1$ <*expression*> | <*expression*> $\texttt{op}_2$ <*expression*>

Pour lever les ambiguités, on distingue plusieurs catégories d'expression. <*expr*$_1$> présente toutes les expressions dont l'opérateur de tête est de précedence au moins celle de $\texttt{op}_1$, de même <*expr*$_2$> présente toutes les expressions dont l'opérateur de tête est de précedence au moins celle de $\texttt{op}_2$, finalement <*expr-simple*> représente les expressions atomiques (en général les constantes, identificateur ou les expressions parenthésées).

On écrit alors la description suivante :

<*expr-simple*> ::= <*constante-entière*> | `"("` <*expression*> `")"`
<*expression*> `::=` <*expr*$_1$>
<*expr*$_1$> `::=` <*expr*$_2$> | <*expr*$_1$> $\texttt{op}_1$ <*expr*$_2$>
<*expr*$_2$> `::=` <*expr-simple*> | <*expr-simple*> $\texttt{op}_2$ <*expr*$_2$>

La symétrie de la définition <*expression*> `::=` <*expression*> `op` <*expression*> qui cause l'ambiguité est cassée en imposant à l'un des deux membres d'être une expression qui ne débute pas par un symbole de même précédence. Le choix du membre gauche ou droit dépend des règles d'associativité souhaitées.

Reconnaissance Lorsque l'on se donne une grammaire (ou un ensemble de définition BNF) et une suite de caractères en entrée, se pose le problème de vérifier si l'entrée appratient bien au langage associé à la grammaire et si oui de construire l'arbre de dérivation syntaxique correspondant.

Il existe deux méthodes principales pour effectuer cette reconnaissance.

Analyse descendante La méthode d'*analyse descendante* consiste à essayer de construire l'arbre de dérivation syntaxique en partant de la racine. À chaque étape, on garde deux objets, une suite de catégories et de caractères qui représente un morceau de l'arbre de dérivation que l'on cherche à construire et une suite de caractères qui correspond à l'entrée. Il y a deux transitions possibles.

1. Si la suite de catégories et de caractères commence par un caractère c
   (a) si l'entrée commence par le même caractère c, on efface ce caractère des deux côtés. Il s'agit d'une action de *lecture*.
   (b) si l'entrée commence par un caractère différent de c, alors il s'agit d'un échec
2. Si la suite de catégories et de caractères commence par une catégorie <*cat*> alors on choisit une des règles de définition associée à <*cat*> et on remplace <*cat*> par le corps de sa définition. Il s'agit d'une action de *d'expansion.*

Évidemment l'expansion peut souvent se faire avec plusieurs définitions possibles, certains des choix peuvent conduire à des échecs ultérieurs dans les étapes de lecture. Le choix de la bonne définition à utiliser peut se faire en utilisant un ou plusieurs caractères de la chaîne d'entrées. Le processus réussit si on arrive dans un état où à la fois la suite de catégories et de caractères et l'entrée sont vides.

Supposons que l'entrée soit `(3+4)`, les étapes de l'analyse descendante sont :

	entrée	action
<*expression*>	`(3+4)`	expansion
`"("` <*expression*> `")"`	`(3+4)`	lecture
<*expression*> `")"`	`3+4)`	expansion
<*expression*> `"+"` <*expression*>`")"`	`3+4)`	expansion
<*constante-entiere*> `"+"` <*expression*>`")"`	`3+4)`	lecture
`"+"` <*expression*>`")"`	`+4)`	lecture
<*expression*>`")"`	`4)`	expansion
<*constante-entiere*> `")"`	`4)`	lecture
`")"`	`)`	lecture
		succès

Analyse ascendante L'*analyse ascendante* procède de manière opposée en essayant de reconstruire l'arbre de dérivation syntaxique en partant des feuilles. L'état est toujours formé d'une suite de catégories et de caractères et d'une entrée. Au départ la suite de catégories et de caractères est vide. Les étapes possibles sont

1. Retirer un caractère de l'entrée et l'ajouter à la suite de catégories et de caractères. C'est une étape de lecture.
2. Si la suite de catégories et de caractères se termine par une séquence qui est la partie droite d'une définition d'une catégorie <*cat*>, remplacer cette sous-séquence par <*cat*>. C'est une étape d'expansions.

Dans certaines situations ont peut au choix effectuer l'une ou l'autre des opérations. Il existe également des situations où deux réductions différentes peuvent avoir lieu. L'analyse réussit si on reste avec une suite de catégories et de caractères réduite à la catégorie initiale pour une entrée vide. Si on reprend notre exemple précédent, on trouve la suite d'étapes suivante :

	entrée	action
	`(3+4)`	lecture
`"("`	`3+4)`	lecture
`"("` <*constante-entière*>	`+4)`	réduction
`"("` <*expression*>	`+4)`	lecture
`"("` <*expression*> `"+"`	`4)`	lecture
`"("` <*expression*> `"+"` <*constante-entière*>	`)`	réduction
`"("` <*expression*> `"+"` <*expression*>	`)`	réduction
`"("` <*expression*>	`)`	lecture
`"("` <*expression*> `")"`		réduction
<*expression*>		succès

L'analyse descendante est relativement simple à mettre en œuvre en particulier en utilisant des procédures récursives pour chaque catégorie. Néanmoins les grammaires usuelles ne présentent pas de bonnes propriétés pour que ce processus soit déterministe. Beaucoup de grammaires permettent de réaliser l'analyse ascendante en regardant uniquement le premier caractère d'entrée pour décider si on doit effectuer une lecture ou une réduction et dans ce dernier cas qu'elle est la réduction à faire. Des outils comme Yacc permettent d'engendrer automatiquement un analyseur ascendant à partir de la description de la grammaire. De plus, il est possible d'associer à chaque règle de la grammaire une instruction qui sera réalisée au moment de la réduction et qui permet ainsi de construire des obkjets en suivante la structure de l'arbre de dérivation syntaxique.

Le résultat de l'analyse syntaxique est souvent un arbre de syntaxe abstraite qui reflète la structure du programme tout en ayant éliminer les aspects superflus de la syntaxe (par exemple les parenthèses dans les expressions qui ne servent qu'à éliminer les ambiguïtés de syntaxe).

3.4 Sémantique

Lorsque la structure du programme est reconnue, il reste encore à lui attribuer un *sens*. La sémantique d'un langage de programmation sert à présenter formellement des relations satisfaites par le programme.

On distingue les sémantiques *statiques* qui établissent des propriétés des programmes en examinant leur corps sans avoir à procéder à l'exécution du programme et les sémantiques *dynamiques* qui formalisent le calcul effectué par le programme.

Sémantique statique Il s'agit le plus souvent des *règles de portées* qui régissent l'ensemble des variables ou fonctions qui sont *visibles* et donc utilisables en un point de programme. Les langages de programmation utilisent également des systèmes de types. Ces deux relations permettent de repérer des erreurs grossières, mais également de calculer des informations utiles (taille des données, nature des variables) nécessaires à la génération de code. Ces vérifications font partie du compilateur.

Sémantique dynamique La sémantique dynamique décrit les étapes du calcul. Cette description n'est pas intégrée au compilateur mais permet de mieux comprendre les calculs correspondant aux constructions d'un programme. Formaliser complètement la sémantique d'un langage est en général fastidueux. Des expériences ont été menées dans le cadre de sous-ensemble de Java utilisés sur des cartes à puces. Cette sémantique serait nécessaire si l'on cherchait à justifier la correction d'un compilateur. La sémantique peut décrire les étapes élémentaires de calcul d'un programme ou au contraire relier un programme au résultat final qui sera calculé.

Notations Une notation usuelle pour définir les sémantiques des langages est d'introduire des relations qui sont définies en utilisant des *règles d'inférence*. Des exemples de relations sont :

- Dans un environnement Γ qui définit la signature des opérations externes et le type des variables, une expression e est bien typée de type τ.
- Dans un environnement Γ qui définit la signature des opérations externes, un programme p est bien formé.
- Dans un environnement Γ qui associe à chaque variable une valeur, le résultat de l'évaluation de l'expression e est v.

Nous nous intéresserons principalement aux règles de portée et de typage. Elles peuvent être décrites par un ensemble de règles d'inférence.

Une règle d'inférence est composée de deux parties :

- Les prémisses de la règle d'inférence sont un ensemble fini éventuellement vide de formules, ou bien encore les relations en cours de définition.
- La conclusion de la règle d'inférence est un cas particulier de la relation que l'on est en train de définir.

Les prémisses et conclusion peuvent comporter des variables. L'interprétation de ces règles d'inférence est que pour toutes les valeurs possibles des variables, si les prémisses sont vérifiées alors il en est de même de la conclusion. De plus cette notation soustend le fait que les relations que l'on définit ainsi sont les plus petites relations qui vérifient cette condition.

On présente un règle d'inférence par une barre de fraction, les prémisses apparaissent au-dessus de la barre et la conclusion en dessous.

$$\frac{\textit{Prémisse}_1 \quad \ldots \quad \textit{Prémisse}_n}{\textit{Conclusion}}$$

Exemple On s'intéresse à un mini langage dans lequel on a des expressions, des instructions et des déclarations.

- Les expressions sont formées à partir des constantes qui peuvent représenter les valeurs booléennes `true` et `false` ou des constantes entières, des variables et des opérations binaires primitives.
- Les instructions sont formées à partir de l'affectation d'une expression à une variable notée $<$*var*$>$`=`$<$*expr*$>$, on peut séquencer plusieurs instructions $<$*instr*$>$`;`$<$*instr*$>$ pour former une nouvelle instruction et également construire des instructions de boucle :
  `while (` $<$*expr*$>$ `) {` $<$*instr*$>$ `}`.
- Les types du langage sont `int` et `bool`.
- Les déclarations sont une suite de couples $<$*type*$>$ $<$*var*$>$ séparées par des point-virgules.
- Les programmes sont composés d'une suite de déclarations et d'une instruction.

Exercice Écrire la syntaxe de ce langage sous forme BNF.

Sémantique statique On définit plusieurs relations décrivant les conditions sous lesquelles un programme est bien formé. Ces relations sont notées

- $\textit{decls} \vdash \textit{expr} : \textit{type}$ qui exprime le fait que l'expression *expr* est bien formée de type *type* dans un environnement comportant les déclarations *decls*.
- $\textit{decls} \vdash \textit{instr}$ **ok** qui exprime le fait que l'instruction *instr* est bien formée dans un environnement comportant les déclarations *decls*.

Ces relations sont définies à l'aide de règle d'inférence, introduites pour chaque construction syntaxique du langage.

$$\frac{}{d \vdash \textit{constante-entière} : \texttt{int}} \qquad \frac{}{d \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

$$\frac{t\ var \in d}{d \vdash var\ :\ t} \qquad \frac{d \vdash e_1 : t_1 \quad d \vdash e_2 : t_2 \quad \mathbf{sign}(\texttt{op}) = (t_1, t_2) \to t}{d \vdash e_1\ \texttt{op}\ e_2\ :\ t}$$

$$\frac{t\ var \in d \quad d \vdash e : t}{d \vdash var\texttt{=}e\ \mathbf{ok}} \qquad \frac{d \vdash i_1\ \mathbf{ok} \quad d \vdash i_2\ \mathbf{ok}}{d \vdash i_1; i_2\ \mathbf{ok}} \qquad \frac{d \vdash e : \texttt{bool} \quad d \vdash i\ \mathbf{ok}}{d \vdash \texttt{while(}e\texttt{)\{}i\texttt{\}}\ \mathbf{ok}}$$

Un programme formé des déclarations d et de l'instruction i sera bien formé si $d \vdash i\ \mathbf{ok}$.

Sémantique dynamique Si on veut modéliser le résultat du programme, on doit encore distinguer le cas des expressions et des instructions. Dans le cas des expressions, la valeur d'un programme pourra être un entier de $\mathbb{Z}$ ou une valeur de vérité que l'on notera `T` ou `F` pour les distinguer de la syntaxe `true` ou `false`. On associe au type `int` le domaine $\mathbb{Z}$ et au type `bool` le domaine formé des deux valeurs {`T`,`F`}. On note $\tilde{t}$ le domaine associé au type t.

Pour définir cette sémantique, on a besoin d'une mémoire qui associe à chacune des variables du programme une valeur, on suppose que le domaine auquel appartient cette valeur correspond au type déclaré de la variable. Il est assez naturel de voir cette mémoire comme une fonction des varaibles du programme dans les valeurs. On a besoin également d'interpréter les opérateurs. À chaque opérateur `op` de la syntaxe on fait correspondre une fonction $\widetilde{\texttt{op}}$ sur les valeurs. On suppose que si la signature de `op` est $(t_1; t_2) \to t$ alors la fonction associée prend en argument des valeurs dans $\widetilde{t_1}$ et $\widetilde{t_2}$ et calcule une valeur dans $\tilde{t}$.

Les instructions définissent une transformation de la mémoire.

On définit les relations :

- $mem \vdash expr \leadsto val$ qui exprime le fait que l'expression *expr* s'évalue en la valeur *val* dans une mémoire *mem*.
- $mem \vdash instr \leadsto mem'$ qui exprime le fait que l'instruction *instr* est bien formée et dans une mémoire *mem* s'évalue en une mémoire *mem'*.

Là aussi on utilise des règles d'inférence pour définir cette sémantique.

$$\frac{\mathbf{val}(constante\text{-}entière) = n}{mem \vdash constante\text{-}entière \leadsto n} \qquad \frac{}{mem \vdash \texttt{true} \leadsto \texttt{T}} \qquad \frac{}{mem \vdash \texttt{false} \leadsto \texttt{F}}$$

$$\frac{mem(var) = v}{mem \vdash var \leadsto v} \qquad \frac{mem \vdash e_1 \leadsto v_1 \quad mem \vdash e_2 \leadsto v_2}{mem \vdash e_1\ \texttt{op}\ e_2 \ \leadsto \widetilde{\texttt{op}}(v_1, v_2)}$$

$$\frac{mem \vdash e \leadsto v \quad mem'(var) = v \quad \forall x \neq var, mem'(x) = mem(x)}{mem \vdash var\texttt{=}e \leadsto mem'}$$

$$\frac{mem_1 \vdash i_1 \leadsto mem_2 \quad mem_2 \vdash i_2 \leadsto mem_3}{mem_1 \vdash i_1; i_2 \leadsto mem_3}$$

$$\frac{mem \vdash e \leadsto \texttt{F}}{mem \vdash \texttt{while(}e\texttt{)\{}i\texttt{\}} \leadsto mem} \qquad \frac{mem \vdash e \leadsto \texttt{T} \quad mem \vdash i; \texttt{while(}e\texttt{)\{}i\texttt{\}} \leadsto mem'}{mem \vdash \texttt{while(}e\texttt{)\{}i\texttt{\}} \leadsto mem'}$$

Il y a dans ce cas une difficulté pour représenter la sémantique d'un programme, en effet, il faut choisir une mémoire initiale dans laquelle évaluer le programme. Or les déclarations ne nous donnent pas forcément de valeurs initiales pour les variables. Des langages comme Java propose une initialisation par défaut.

Les règles de sémantique statique ou dynamique peuvent se traduire sous forme de programme pour réaliser le typage ou bien un évaluateur de programme.

3.5 Conclusion

Dans ce chapitre nous avons évoqué rapidement les différentes phases de la compilation. Nous avons plus particulièrement expliqué les notations permettant de décrire précisément un langage de programmation. Comprendre ces notations permet d'appréhender de nouveaux langages de programmation et de comprendre les erreurs détectées par le compilateur. Les techniques mises en œuvre dans l'analyse lexicale et l'analyse syntaxique sont utiles dans de nombreuses situations où on doit transformer des données. De nombreux outils spécialisés facilitent la construction de programmes d'analyse.

Chapitre 4

Langages

L'objectif de ce chapitre est de donner les éléments essentiels de la programmation. Ceux-ci seront illustrés principalement à l'aide du langage C [3]. Des exemples seront également donnés dans le langage Fortran [5, 6, 4]. La connaissance de plusieurs langages permet d'identifier les concepts fondamentaux et prépare à l'apprentissage des différents langages de programmation utilisés. A travers ces langages, il sera mis l'accent sur l'acquisition de bons principes de programmation largement indépendants du langage utilisé [2]. Nous ne décrirons pas l'ensemble des constructions de C de manière exhaustive mais introduirons les constructions les plus typiques.

4.1 A propos des langages Fortran et C

Le langage C a été créé à l'origine par Dennis Ritchie comme langage pour la programmation du système Unix. Il est largement répandu comme un langage généraliste, il permet une programmation proche de la machine tout en offrant des structures de données et des structures de contrôle de haut niveau. La plupart des architectures disposent d'un compilateur du langage C, c'est donc un langage largement répandu pour la programmation de systèmes embarqués.

Fortran est un langage ancien, il a été développé dans l'équipe de J. Backus à IBM à partir de 1957. L'objectif était d'avoir un langage aussi efficace que l'assembleur mais qui soit lisible pour un humain et qui permette d'écrire de manière naturelle des formules mathématiques. Le nom FORTRAN vient de **FOR**mula **TRAN**slation. L'usage de Fortran s'est répandu et de nombreux dialectes existent. Des efforts de standardisation ont été réalisés, en 1966 d'abord puis en 1977, 1990 et 1995. Dans ce cours, nous illustrerons principalement le langage FORTRAN par sa version Fortran 77.

4.2 Les constructions élémentaires

4.2.1 Les unités lexicales

Pour décrire les unités lexicales, nous utiliserons les abréviations suivantes :

- *chiffre* représente un chiffre de `0` à `9`
- *alpha* représente une lettre minuscule ou majuscule de `a` à `z` ou de `A` à `Z`
- Les commentaires débutent par les caractères `/*` et se terminent à la première occurrence de `*/`. Ils sont considérés comme des espacements, de même que les blancs, tabulations, retour à la ligne, saut de page ...Ils servent à séparer des unités lexicales et sont ignorés dans le reste de l'analyse.
- Les identificateurs sont une séquence de lettres (de `a` à `z` et de `A` à `Z` ainsi que le caractère de soulignement `_`) et de chiffres et commencent par une lettre.

  L'expression régulière correspondante est : $(\mathit{alpha}|\_)(\mathit{alpha}|\_|\mathit{chiffre})^*$.
  Les majuscules sont significatives.
- Il y a une liste de mots-clés qui sont des identificateurs réservés qui ne peuvent être utilisés pour un autre usage. Par exemple **while**. Dans ce cours, nous écrirons les mots-clés en caractères gras.
- Les constantes peuvent être des constantes entières, des constantes flottantes, des caractères ou des constantes énumérées. Pour l'instant nous ne considérons que des cas particuliers de constantes.
  - Une constante entière est une suite de chiffres qui ne commence pas par 0
  - Une constante caractère comporte un caractère entre apostrophe. Il y a des conventions pour représenter des caractères spéciaux. Ceux-si sont introduits par le caractère de contre-oblique (backslash) suivi d'un caractère particulier. On a par exemple :

`\n`	passage à la ligne
`\'`	le caractère `'`
`\\`	le caractère `\`
`\"`	le caractère `"`
`\b`	retour en arrière (backspace)
`\f`	saut de page

  - Les constantes énumérées sont des identificateurs déclarés dans des types énumérés (cf 4.5.1)
  - Les constantes flottantes ont une forme générale correspondant à l'expression régulière :

    $$\mathit{chiffre}^{+}\,.\,\mathit{chiffre}^{*}((e|E)(-)?\mathit{chiffre}+)?$$

    On peut aussi ne pas mettre de chiffres avant le point, mais il faut alors au moins un chiffre après le point :

    $$.\,\mathit{chiffre}^{+}((e|E)(-)?\mathit{chiffre}+)?$$

    On peut également ne pas mettre de point mais il faut alors obligatoirement un exposant :

    $$\mathit{chiffre}^{+}(e|E)(-)?\mathit{chiffre}+$$

  - Les constantes de chaînes sont des séquences de caractères entre guillemets. Elles ne peuvent contenir de retour à la ligne ou de guillemets qui doivent être représentés à l'aide de caractères spéciaux.

4.2.2 Les types de base

Les types de base de C que nous utiliserons dans cette partie sont **int** (les entiers), **char** (les caractères, codés sur un octet, c'est-à-dire 8 bits), **float** (les flottants).

Le langage C propose des types différents pour représenter des entiers dans différents intervalles.

À l'aide de p bits, on peut représenter $2^p - 1$ nombres. On peut choisir de représenter les nombres de 0 à $2^p - 1$, on parle alors d'entier non signé, ou bien de représenter les nombres de -2^{p-1} à $2^{p-1} - 1$, on parle alors d'entier signé.

Le langage propose les types **short int** (aussi noté **short**) et **long int** (aussi noté **long**) qui sont des entiers signés qui sont codés sur un nombre de bits dépendant de l'architecture. On peut utiliser le mot-clé **unsigned** pour indiquer que les entiers sont interprétés de manière non signée et que le calcul se fera alors modulo 2^p où p est le nombre de bits.

Les types **int** et ses dérivés, le type **char** ou bien les types énumérés sont tous considérés comme des types *entiers* et on pourra convertir les valeurs entre ces différents types. Certaines conversions se feront de manière automatique au moment des opérations. Le principe général est que lorsque l'on doit effectuer une opération entre deux nombres dans des types différents, on va s'aligner sur celui qui présente la plus grande précision. Si à un endroit où on attend une expression de type **int**, on met une expression dans un type *plus petit* comme un **short**, un **char** ou un objet dans un type énuméré alors celui-ci sera considéré comme un **int**.

De même, il existe plusieurs types flottants, les **double** sont des flottants en double précision et on a même des **long double** en précision étendue.

On peut convertir un entier en flottant (attention un entier n'est pas forcément représentable de manière exacte comme un flottant) et un flottant en entier (en tronquant la partie fractionnaire).

Booléens Il n'y a pas de type booléen spécifique dans le langage C. Les conditions seront représentées par un entier. L'entier 0 correspond à la valeur de vérité `faux` et les entiers non-nuls sont interprétés comme la valeur de vérité `vrai`.

4.2.3 Expressions

Expressions simples

Les expressions élémentaires sont les identificateurs, les constantes ou bien les expressions quelconques entre parenthèses

<expr-elem> `::=` *<constante>* | *<ident>* | `"("` *<expr>* `")"`

Les expressions peuvent être formées en composant les expressions à l'aide d'opérateurs préfixe, infixe (on verra plus tard les opérateurs post-fixes)

<expr> `::=` *<expr-elem>*
<expr> `::=` *<op-prefix>* *<expr>*
<expr> `::=` *<expr>* *<op-infix>* *<expr>*

Parmi les opérateurs on trouve évidemment les opérateurs arithmétiques et relationnels :

<op-prefix>	`::=`	`"-"`	moins unaire
	|	`"!"`	négation logique
<op-infix>	`::=`	`"+"` | `"-"` | `"*"`	opérateurs arithmétiques
	|	`"/"` | `"%"`	division et reste modulo
	|	`"=="` | `"!="`	égalité, différence
	|	`"<"` | `">"` | `">="` | `"<="`	opérateurs de comparaison
	|	`"&&"` | `"||"`	opérateurs logiques: `et`, `ou`

Appel de fonctions

Une expression peut comporter un appel de fonction. La fonction peut être appliquée à des arguments. La syntaxe est :

<expr> `::=` *<ident>* `"("` [*<suite-expr>*] `")"`
<suite-expr> `::=` *<expr>* { `","` *<expr>* }

La fonction peut être définie par l'utilisateur dans son programme ou faire partie d'un autre module de l'utilisateur ou d'une bibliothèque. Si la fonction n'est pas définie dans le module, son *profil* doit apparaître. Nous reviendrons sur ce point dans la partie concernant les déclarations.

La fonction `printf` La fonction `printf` est largement utilisée dans les programmes même élémentaires. Nous reviendrons plus tard sur ces caractéristiques, cependant il est important d'en comprendre le fonctionnement de base.

La commande `printf` est déclarée dans la bibliothèque `<stdio.h>` qu'il convient d'inclure par la directive `#include <stdio.h>`.

La commande `printf` permet d'afficher sur la sortie standard (l'écran) des valeurs dans différents types : chaînes de caractères, entiers, flottants, caractères ...C'est une commande qui prend en argument un nombre variable d'expressions. Le premier argument décrit le format d'impression et conditionne le nombre et le type des autres arguments.

Un *format* est une chaîne de caractères entre guillemets qui contient des caractères ordinaires et des commandes de conversion. Les commandes de conversion sont :

%s	affichage d'une chaîne
%d, %i	affichage d'un entier
%f	affichage d'un flottant
%c	affichage d'un caractère
%%	affiche le caractère %

La fonction `printf` va prendre autant d'arguments que de commandes de conversion dans le format. La chaîne va être affichée en remplaçant chaque commande de conversion par l'affichage de la valeur de l'expression correspondante.

Si on veut afficher deux variables x et y de type chaîne de caractères et entier. On pourra utiliser la commande : `printf`("%s%d",x,y)

On peut insérer les commandes de conversion au milieu d'une chaîne, par exemple en écrivant : `printf`("Personne %s née en %d\n",x,y). Si x est une variable qui vaut `"X"` et y vaut 2000 alors la commande affichera

Personne X née en 2000

On peut ajouter dans les commandes de conversion des indications de formatage entre le caractère % et le caractère de conversion. Ainsi `%6d` affiche un entier sur au moins 6 caractères ou bien `%3.1f` affiche un flottant sur au moins 3 caractères avec un maximum d'un caractère après la virgule.

Variables et affectation

Un programme agit en transformant l'état de la mémoire. On lit ou modifie les valeurs stockées en mémoire par l'intermédiaire de variables. Une variable est un identificateur déclaré dans le programme et qui va être associé à une portion de la mémoire. Une variable joue un double rôle, elle représente un emplacement en mémoire (une adresse, aussi appelée la valeur gauche de la variable) mais également la valeur stockée à cet emplacement (qui va être du type déclaré de la variable, on parle aussi de valeur droite de la variable). L'utilisation de l'une ou l'autre des interprétations dépend du contexte d'utilisation de la variable.

L'opération de base est l'affectation d'une valeur à une variable. L'affectation se fait de manière générale vers toute expression à laquelle on peut associer un emplacement mémoire, c'est-à-dire une valeur gauche. Pour l'instant nous ne considérons que les variables.

<*val-gauche*> `::=` <*ident*>
<*expr*> `::=` <*val-gauche*> `=` <*expr*>

Attention à ne pas confondre le signe `==` qui est l'opérateur binaire qui teste l'égalité de deux expressions et le signe d'affectation `=`. En C, l'affectation est une expression et non pas une instruction.

Cette expression est licite lorsque la variable v et l'expression e sont dans des types numériques. L'expression e est convertie dans le type de la variable v. L'expression $v = e$ a pour type le type de v et pour valeur la valeur de v après affectation.

Abréviations Le langage C offre beaucoup de notations raccourcies pour représenter des affectations qui sont assez largement utilisées à défaut d'être très lisibles.

Ainsi l'opérateur d'affectation peut être précédé d'un opérateur binaire infixe `+`, `-`, `*`, `/`, `%`. La signification de l'expression v+=e est v=v+e et de même pour les autres opérateurs.

On peut également incrémenter ou décrémenter une variable. Cela se fait à l'aide des opérateurs `++` et . On peut appliquer ces opérateurs à des valeurs gauches, ils ont pour effet d'incrémenter ou de décrémenter la valeur gauche.

Ainsi ++v est équivalent à v+=1 qui est lui-même équivalent à v=$v+1$. De même `--`v est équivalent à v`-=`1 qui est lui-même équivalent à v=$v-1$. Une variante postfixe des mêmes opérateurs existe, elle a le même effet sur la mémoire mais renvoie la valeur de son opérande, ie la valeur de la variable avant la décrémentation ou l'incrémentation.

Déclarations Les variables doivent être déclarées avant d'être utilisées. Une déclaration est formée d'un type et d'une suite d'identificateurs séparés par des virgules et est terminé par un point virgule.

<declaration> `::=` *<type>* *<ident>* { `","` *<ident>* } `";"`

Il est possible d'initialiser les variables au moment de la déclaration, la syntaxe générale des déclarations est donc :

<declaration> `::=` *<type>* *<ident>* [`"="` *<expr>*] { `","` *<ident>* [`"="` *<expr>*] } `";"`

4.2.4 Instructions

L'instruction de base est formée d'une expression suivie d'un point-virgule. On peut également former des instructions en mettant plusieurs instructions dans un bloc entre accolades. Ce bloc peut contenir des déclarations de variables. Les instructions peuvent également être des conditionnelles ou bien des itérations.

<instruction> `::=` [*<expr>*] `";"`
<bloc> `::=` `"{"` {*<declaration>*} {*<instruction>*} `"}"`
<instruction> `::=` *<bloc>*
<instruction> `::=` **if** `"("` *<expr>* `")"` *<instruction>* [**else** *<instruction>*]
<instruction> `::=` **while** `"("` *<expr>* `")"` *<instruction>*
<instruction> `::=` **for** `"("` [*<expr>*] `";"` [*<expr>*] `";"` [*<expr>*] `")"` *<instruction>*

Remarque : La boucle **for** est en fait du sucre syntaxique pour une boucle **while** dans laquelle on met en évidence une étape de prétraitement avant l'entrée de boucle et une étape systématique de post-traitement à la fin de chaque passage dans la boucle.

On codera par exemple une itération pour un indice i allant de 0 à n par :

```
for (i = 0; i <= n; i++) ...
```

L'instruction **for** (e_1;e_2;e_3) i est équivalente au programme : e_1 ; **while** (e_2) {i e_3;}.
Les expressions dans la boucle **for** sont optionnelles, si e_2 est ommis alors on considère que la condition est toujours vrai, on a donc une boucle infinie dont la sortie sera gérée par une mécanisme d'échappement.

4.2.5 Structure du programme

Un programme est une suite de déclarations. On a déjà vu les déclarations de variables. On peut également mettre des déclarations de fonctions. Une déclaration de fonction débute par le type des valeurs calculées par les fonctions puis le nom de la fonction ainsi que ses arguments. Le corps de la fonction est formée d’un bloc d’instructions.

<declaration> `::=` *<type>* *<ident>* `"("` [*<arguments>*] `")"` *<bloc>*
<arguments> `::=` *<type>* *<ident>* { `","` *<type>* *ident* }

Le programme doit forcément comporter une fonction dont le nom est `main`. Cette fonction sera le point d’entrée du programme. C’est-à-dire que c’est cette fonction qui sera appelée au lancement de l’exécutable.

4.2.6 Le cas du Fortran

Format de fichier FORTRAN est un langage ancien, longtemps utilisé sur des cartes perforées. . .Il a gardé des contraintes syntaxiques fortes sur le format des programmes. Un programme FORTRAN 77 devait être écrit en majuscules, les lignes doivent faire exactement 72 caractères. Chaque ligne comportait essentiellement une instruction. Les 5 premiers caractères étaient utilisés pour étiqueter les instructions (en vue d’un saut) le 6ème caractère était utilisé pour indiquer si la ligne courante est la suite de la ligne précédente, les commentaires ne pouvaient être mis que sur des lignes complètes. Ces restrictions ce sont relachées avec la norme Fortran 90, et heureusement les compilateurs actuels dont `g77` adoptent une vision plus moderne de la syntaxe. Néanmoins, FORTRAN a des particularités syntaxiques surprenantes par rapport aux langages de programmation modernes.

- Un commentaire débute par le caractère `!` et se termine à la fin de la ligne
- On met en général une instruction par ligne, néanmoins on peut mettre plusieurs instructions sur une ligne en les séparant par le caractère `";"`. Si une instruction est trop longue pour tenir sur une ligne, on peut couper la ligne en mettant le caractère `"&"` à la fin. Dans le cas où on coupe une chaîne de caractères, on met également un `"&"` sur la ligne suivante au début de la suite de la chaîne.
- Les majuscules ne sont pas significatives (c’est-à-dire que le compilateur ne distingue pas deux identificateurs qui ont les mêmes lettres mais qui diffèrent par l’usage des majuscules).

Constantes de chaînes Les constantes de chaînes sont formées d’une suite de caractères entourées d’apostrophes `’` ou bien de guillemets `"`. Pour inclure un guillemet ou une apostrophe au milieu d’une chaîne délimitée par le même caractère, il suffit de le doubler. Ainsi pour représenter la chaîne `J’ai dit : "Bonjour"` on pourra utiliser l’une des deux notations suivantes :

- `"J’ai dit : ""Bonjour!"""`
- `’J’’ai dit : "Bonjour!"’`

Variables FORTRAN accepte de manipuler des variables qui ne sont pas déclarées, le type de telles variables est déterminé par sa première lettre. Cette pratique peut s’avérer très dangereuse car elle empêche le compilateur de détecter des erreurs dans l’écriture des noms de variable. Aussi il est fortement recommandé de toujours insérer la ligne suivante qui interdit l’utilisation de variables non déclarées.

```
IMPLICIT NONE
```

Types Le langage FORTRAN distingue essentiellement trois types de données.
Le type `CHARACTER` pour représenter des caractères (qui sont des chaînes réduites à un caractère), le type `INTEGER` pour les entiers, `REAL` pour les flottants, `LOGICAL` pour les valeurs booléennes `.TRUE.` et `.FALSE.`.

Expressions La construction des expressions en FORTRAN est relativement standard. La seule particularité est que les opérateurs arithmétiques ont aussi une notation non symbolique, les opérateurs logiques n'ont qu'une notation textuelle.

`.GT.`	`>`	`.GE.`	`>=`	`.AND.`
`.LT.`	`<`	`.LE.`	`<=`	`.OR.`
`.EQ.`	`==`	`.NE.`	`/=`	`.NOT.`

La commande `PRINT` est d'un usage simple, le premier argument indique la sortie sur laquelle imprimer, on utilise `*` pour la sortie standard (l'écran); les autres arguments séparés des virgules sont imprimés successivement. Le type de ces arguments peut être quelconque, la conversion vers une chaîne se fait automatiquement.

Instruction L'affectation d'une valeur à une variable s'écrit comme en C à l'aide du signe `=`. Il y a plusieurs formes syntaxiques pour les conditionnelles. Les instructions d'itération sont des boucles paramétrées par un indice variant entre deux bornes avec un pas variable. Il y a également une instruction de boucle non conditionnelle : une instruction spéciale `EXIT` peut être utilisée dans le corps de la boucle pour sortir de l'instruction.

```
<instruction> ::= <ident> "=" <expr>
<instruction> ::= IF "(" <expr> ")" <instruction>
<instruction> ::= IF "(" <expr> ")" THEN
                      <instruction> { <instruction> }
                  { ELSEIF "(" <expr> ")" THEN
                      <instruction> { <instruction> } }
                  [ ELSE
                      <instruction> { <instruction> } ]
                  END IF
<instruction> ::= DO
                      <instruction>
                  END DO
<instruction> ::= DO <ident> "=" <expr> "," <expr> ["," <expr>]
                    <instruction>
                  END DO
```

Structure d'un programme Un code est entouré de déclarations de début et de fin de programme.

```
PROGRAM <ident>
...
END PROGRAM <ident>
```

Une déclaration est comme en C formée d'un type et d'une ou plusieurs variables séparées par des virgules. Le signe `::` est optionnel entre le type et les variables et devient obligatoire si on veut initialiser les variables.

```
<declaration> ::= <type> ["::"] <ident> { "," <ident> }
```

4.2.7 Autres constructions élémentaires

Tableaux simples

Nous reviendrons dans la partie 4.4.2 sur les manipulations de tableaux. Nous présentons ici la forme simple d'un tableau uni-dimensionnel de taille connue statiquement.

Une variable t représentant un tableau de taille 10 d'éléments de type **int** se déclare de la manière suivante :

```
int t[10];
```

La syntaxe des déclarations est étendue de la manière suivante :

<declaration> `::=` *<type>* *<ident>*`"["` *<constante>* `"]"`

Les tableaux du langage C sont indicés à partir de 0. Ainsi les valeurs du tableau t sont rangées de $t[0]$ à $t[9]$. Chacune de ces cases correspond à un morceau de la mémoire qui peut être modifié, ces expressions sont donc des valeurs gauches. On ajoute donc une nouvelle définition pour les valeurs gauches (qui peuvent se trouver à gauche d'une affectation) et une définition générale qui dit que toute valeur gauche peut également être interprétée comme expression (en regardant la valeur stockée à l'emplacement mémoire).

<val-gauche> `::=` *<ident>* `"["` *<expr>* `"]"`
<expr> `::=` *<val-gauche>*

Les tableaux simples en Fortran Dans le langage Fortran, un tableau uni-dimensionnel peut être déclaré en spécifiant l'indice le plus petit et l'indice le plus grand :

<declaration> `::=` *<type>* [`"::"`] *<ident>*`"("` [*<constante>* `":"`] *<constante>*`")"`

L'indice bas par défaut est 1. Ainsi la déclaration :

```
INTEGER t(10)
```

est équivalente à

```
INTEGER t(1:10)
```

et les cases de ce tableau seront repérées de $t(1)$ à $t(10)$.

Expressions conditionnelles

On a vu les instructions conditionnelles. Le langage C propose également une syntaxe pour des expressions conditionnelles :

<expr> `::=` *<expr>* `"?"` *<expr>* `":"` *<expr>*

Ainsi l'expression `a < b? a : b` représente le minimum de `a` et `b`.

Dans l'expression e_1 `?` e_2 `:` e_3, les types de e_2 et e_3 peuvent être différents (un entier et un flottant par exemple). Le type de l'expression conditionnelle sera le plus petit type compatible avec les deux expressions et une conversion aura éventuellement lieu.

4.3 Fonctions et procédures

Les fonctions sont essentielles pour structurer le programme en mettant en évidence le découpage d'une tâche en sous-tâches plus simples. Les fonctions permettent également de factoriser des morceaux de code. L'utilisation de fonctions récursives fournit des méthodes très élégantes pour la résolution de certains problèmes.

4.3.1 Déclaration

La déclaration d'une fonction est composée de la *signature* (on dit aussi le *prototype*) de la fonction qui déclare le type de retour de la fonction, le nom de la fonction et le nom et le type de ses *paramètres* (on parle aussi d'*arguments formels*). Vient ensuite le corps de la fonction qui a la forme d'un bloc, c'est-à-dire qu'il contient une suite de déclarations de variables et une suite d'instructions. Contrairement au langage Pascal, il n'est pas possible d'imbriquer les déclarations de fonctions.

Un exemble de fonction calculant la factorielle ($n! = n \times (n-1) \times \cdots \times 1$) est :

```
int fact (int n) {
   int x = 1, i;
   for (i=2; i <= n; i++) x*=i; printf("fact (%d) = %d \n",n,x);
}
```

Les paramètres de la fonction (comme n dans l'exemple) sont des variables *muettes* (on dit aussi *liées*) c'est-à-dire que l'on ne change rien au programme si on remplace dans la signature et le corps de la fonction ces noms par d'autres qui ne sont pas utilisés par ailleurs dans le programme.

4.3.2 Valeur de retour

L'appel de fonction se fait en donnant des valeurs aux paramètres de la fonction (on parle d'*arguments effectifs*). L'effet de cet appel est d'affecter aux paramètres de la fonction les valeurs des arguments effectifs puis d'exécuter le corps de la fonction.

Dans les langages ou les notions d'expressions et d'instructions sont clairement différenciées, on distingue les procédures et les fonctions. L'appel d'une procédure est une instruction, elle agit par effet de bord, l'appel de fonction est une expression à laquelle est associée une valeur (qui peut également agir par effet de bord). En C, même si les expressions et les instructions sont distinguées, les expressions sont simplement transformables en instructions via le signe `";"`. Il n'y a donc pas de raison de distinguer fonction et procédure. Néanmoins le type **void** (qui signifie vide) permet d'expliciter le fait que la fonction ne renvoie aucune valeur de retour pertinente.

Le résultat de l'appel de fonction sur des arguments est une expression. Une fonction peut agir par effet de bord, elle associe également une valeur résultat de l'appel de fonction (on parle de *valeur de retour* de la fonction). Cette valeur peut être utilisée par le programme appelant.

Pour déterminer cette valeur, on utilise une instruction spéciale **return** à l'intérieur du bloc qui définit la fonctions.

<instruction> `::=` **return** [*<expr>*] `";"`

Lorsque l'instruction **return** est exécutée, elle provoque la fin de l'exécution du corps de la fonction. L'exécution se poursuit à l'instruction qui suivait l'appel de fonction. La valeur de l'expression associée au **return** est prise comme valeur de l'expression d'appel.

Ainsi une meilleure manière d'écrire la fonction factorielle est :

```
int fact (int n) {
   int x = 1, i;
   for (i=2; i <= n; i++) x*=i; return(x);
}
```

L'expression dans l'instruction **return** est optionnelle, la fonction peut ne pas retourner de valeur utile. Une instruction **return ;** sans argument est implicitement placée à la fin du bloc de définition de la fonction.

Le cas de la fonction `main` Une fonction `main` est obligatoire dans le programme comme point d'entrée de l'exécution. On convient en général que cette fonction retournera l'entier 0 à l'environnement appelant lorsque l'exécution s'est passée dans problème et des valeurs non nulles pour signaler différents types d'erreur.

4.3.3 Variables

Passage de paramètres

Le passage des arguments des fonctions en C se fait toujours *par valeurs*. Cela signifie que si la déclaration de la fonction est de la forme

type nom (t_1 $x\_1,\ldots,t\_n$ x_n) `{` ***decls instrs*** `}`

et si l'appel est de la forme : *nom* $(e_1,\ldots,e_n)$ alors tout se passera comme si on effectuait les opérations suivantes (en supposant que les noms x_i ne sont pas utilisés par ailleurs) :

`{`t_1 $x\_1 = e\_1;\ldots$ t_n $x\_n = e\_n;$ ***decls instrs*** `}`

Les variables $x\_1 = e\_1;\ldots x\_n = e\_n$ sont nouvellement créées et la place mémoire ainsi mobilisée sera libérée à la fin de l'exécution du corps de la fonction. En particulier, toute modification effectuée sur les variables `x_i` dans le corps de la fonction n'aura pas d'incidence sur les variables du programme appelant.

On verra dans la section 4.4 que la possibilité de passer par valeur un pointeur représentant l'adresse d'une variable permet d'écrire des fonctions pouvant modifier des variables globales passées en argument.

Visibilité

Redéfinition Dans une suite de déclarations de variables (en début de programmes, dans les paramètres d'une fonction ou au début d'un bloc). il n'est pas possible d'introduire deux fois le même nom. Par contre, il est possible de réutiliser les mêmes noms dans deux suites ayant des rôles différent. Par exemple, le programme suivant produit des avertissements mais n'échoue pas.

```
int a = 1;
int f (int a) { int a = 2; return (a);}
int main () {
  int a = 3;
  printf("fonction %d \n", f(a));
}
```

Chaque nouvelle déclaration de `a` a pour effet de réserver un emplacement mémoire et de masquer les définitions précédentes de `a` dans la partie du programme où cette déclaration est visible. Il peut s'agir de la suite du fichier dans le cas de variables globales, ou bien du bloc de déclarations ou encore du corps de la fonction dans le cas des paramètres.

Variables visibles dans le corps d'une fonction Dans le corps d'une fonction, on peut utiliser les paramètres de la fonction, les variables apparaissant dans les déclarations du bloc de définition de la fonction mais aussi les variables *globales* (aussi dites *externes*) déclarées en dehors de toute fonction. Si ces variables sont déclarées dans un autre fichier que la fonction ou après la fonction qui les utilise, alors il est nécessaire de rappeler leur type par une déclaration :

```
  extern <type> <ident> { "," <ident> } ";"
```

Cette déclaration peut être intégrée aux déclarations locales dans le bloc de définition de la fonction ou bien se trouver en dehors de toute fonction, auquel cas les variables ainsi déclarées peuvent être utilisées dans toutes les fonctions.

De même l'utilisation d'une fonction définie en dehors du fichier, ou bien après la fonction qui l'utilise. requiert la déclaration préalable de son prototype :

```
  extern <type> <ident>  "(" [<arguments>] ")" ";"
```

Attention, en l'absence d'un prototype déclaré pour une fonction utilisée, le compilateur attribuera un prototype par défaut. La valeur de retour sera arbitrairement de type **int**, ce qui peut conduire à des erreurs délicates.

Les déclarations de prototypes et de variables externes sont souvent regroupées dans un fichier d'en-tête (*header*) qui porte traditionnellement le suffixe `.h` et que l'on inclut en général à l'aide d'une directive

```
#include <decls.h>
```

Masquer des noms de variables ou de fonctions Il peut être intéressant de restreindre l'accès à certaines variables ou fonctions en dehors du fichier où elles sont définies. Ainsi on peut rendre l'implantation d'une structure de données abstraite, ce qui améliore la robustesse des programmes qui utilisent cette structure. Pour cela il suffit d'ajouter l'étiquette **static** avant le type de la variable ou le type de retour de la fonction lors de sa déclaration.

L'attribut **static** peut également s'appliquer à des variables locales dans le corps d'une fonction. Ces variables se comporteront comme des variables locales, en particulier leur valeur est conservée entre deux appels de la fonction. Cependant la visibilité de ces variables est réduite au corps de la fonction où elle est introduite.

4.3.4 Fonctions récursives

Il est possible d'utiliser la fonction que l'on est en train de définir dans le corps de celle-ci. C'est ce que l'on appelle une fonction récursive. À chaque appel de fonction, une nouvelle partie de la mémoire est réservée pour les paramètres et les variables globales. Les fonctions récursives permettent ainsi de réserver une quantité de mémoire non bornée qui ne sera rendue qu'à la fin de l'ensemble des appels.

Ainsi, on peut écrire la fonction factorielle de manière récursive :

```
int fact (int n) {
 if (n<=1) return 1; else return (n * fact (n-1));
}
```

Du point de vue opérationnel, les valeurs successives de la variable n vont s'empiler jusqu'au cas $n = 1$ qui va déclencher la suite de multiplications jusqu'à obtenir le résultat.

Récursion terminale Certaines fonctions récursives ne laissent pas de calcul en attente au moment de l'appel récursif, on dit qu'elles sont *récursives terminales*. Les bons compilateurs engendrent un code itératif pour de telles fonctions récursives qui s'exécute dans un espace mémoire borné.

Ainsi, on peut une fois encore modifier la définition de la fonction factorielle, en utilisant un argument supplémentaire qui sert *d'accumulateur*.

```
int fact_acc (int n,int p) {
  if (n <= 1) return p; else return fact_acc(n-1,n*p);
```

```
}

int fact (int n) { fact_acc(n,1); }
```

4.3.5 Le cas de Fortran

En Fortran, on peut définir des sous-routines et des fonctions. Les sous-routines et les fonctions sont des blocs indépendants du programme principal.

La syntaxe d'une sous-routine est :

`SUBROUTINE` *<ident>* `"("` *<ident>* {`","` *<ident>*} `")"`
  {*<declaration>* }
  {*<instruction>* }
`END SUBROUTINE` *<ident>*

Une particularité est que les paramètres formels doivent être déclarés avec leur type dans les déclarations de la sous-routine. Une instruction `RETURN` peut être placée en un point de programme, elle provoque la sortie de la routine. Elle est implicitement positionné avant la fin de la routine.

L'appel à une sous-routine se fait à l'aide de la commande `CALL`.

<instruction> `::=` `CALL` *<ident>* `"("` *<expr>* {`","` *<expr>*} `")"`

Le cas des fonctions La déclaration d'une fonction se fait de la manière suivante, la seule différence avec les sous-routines est la présence d'une valeur de retour :

<type> `FUNCTION` *<ident>* `"("` *<ident>* {`","` *<ident>*} `")"`
  {*<declaration>* }
  {*<instruction>* }
`END FUNCTION` *<ident>*

Les fonctions obéissent aux mêmes règles (déclaration de paramètres, `RETURN`) que les sous-routines. Afin d'attribuer une valeur de sortie à la fonction on utilise une affectation à une vatiable portant le même nom que la fonction.

Lors de l'appel de la fonction, il doit exister une variable de même nom que la fonction et qui a pour type le type des valeurs de retour de la fonction.

4.3.6 Mode de passage de paramètres

On a vu que dans le langage C, les paramètres étaient passés *par valeur*. C'est-à-dire que de nouvelles variables sont déclarées et initialisées avec les valeurs des expressions passées en argument.

Dans le langage Fortran, le passage des paramètres se fait *par référence*. C'est-à-dire que dans le corps de la fonction les paramètres formels font directement référence aux valeurs gauches des arguments (lorsque c'est possible). En particulier des modifications apportées aux paramètres formels peuvent affecter les valeurs des variables utilisées en argument.

Considérons le même programme écrit en C et en Fortran :

```
int f(int z) { z += 2; }
int main() {int x = 3; printf("f(x)=%d, pour x=%d\n",f(x),x);}
```

La valeur de `x` n'est pas modifiée par l'appel de fonctions, on voit bien s'afficher :

```
f(x)=5, pour x=3
```

L'analogue en Fortran s'écrit :

```
PROGRAM main
IMPLICIT NONE
INTEGER x, f
x = 3
PRINT*,"f(x)=",f(x)," pour x=",x
END PROGRAM main

INTEGER FUNCTION f(z)
INTEGER z
z=z+2
f=z
END FUNCTION f
```

Le résultat est différent, on obtient :

```
 f(x)= 5 pour x= 5
```

Le calcul de `f(x)` a eu pour effet de modifier la valeur de `x`. La tentative de calcul de `f` sur une valeur ne correspondant pas à une variable produit une erreur à l'exécution.

Le passage par référence est utile lorsque l'on cherche à programmer des fonctions qui agissent de manière paramétrique sur des variables.

Par exemple, on peut écrire en Fortran une sous-routine qui échange les valeurs de deux variables :

```
PROGRAM main
IMPLICIT NONE
INTEGER x,y
x = 3; y = 5
CALL echange(x,y)
PRINT*,"x=",x," y=",y
END PROGRAM main

SUBROUTINE echange(x,y)
INTEGER x,y,tmp
tmp = y; y = x; x = tmp
END SUBROUTINE echange
```

On obtient bien le résultat :

```
 x= 5 y= 3
```

La tentative de faire naivement la même chose en C donne :

```
int echange(int x, int y) int tmp = y; y = x; x = tmp;

int main()
  int x = 3, y = 5;
  echange(x,y);
  printf("x=%d, y=%d\n",x,y);
```

et produit l'affichage suivant, qui montre que les valeurs de `x` et `y` restent inchangées.

```
x=3, y=5
```

A propos des macros Il existe d'autres modes de passage de paramètres. Par exemple, le préprocesseur du langage C permet la définition de macros paramétrées. Les macros permettent de simuler des fonctions dans le code source tout en évitant le coût de l'appel de fonction à l'exécution. De plus les macros n'ont pas de prototype, on peut ainsi factoriser un code qui sera utilisé sur différentes structures de données. Lors du passage du pré-processeur, l'appel de macro est remplacé par le corps de la macro dans lequel chaque référence à un paramètre est textuellement remplacé par l'argument correspondant.

Supposons que l'on définisse une macro et une fonction calculant le max de deux nombres en utilisant des opérations arithmétiques. On peut s'attendre à ce que ces deux définitions renvoient les même résultats.

```
#define maxm (a,b) ( abs(a - b) + b + a ) / 2
int max (int a, int b) { return ( abs(a - b) + b + a ) / 2; }
int main () {
  int x = 1, y = 2;
  printf("fonction %d macro %d \n", max(x+y,x-y), maxm(x+y,x-y));
}
```

Or on obtient le résultat suivant :

```
fonction 3  macro 1
```

L'option `gcc -E` permet d'afficher le source du programme après l'étape de pré-processing. On a en fait exécuté la commande suivante :

```
printf("...", max(x+y,x-y), ( abs( x+y  -  x-y ) +  x-y  +  x+y  ) /2 );
```

On remarque que le remplacement textuel des paramètres par les arguments dans les macros a eu pour effet de dupliquer les expressions `x+y` et `x-y`, ce qui duplique inutilement un calcul. D'autre part l'absence de parenthésage autour de `x-y` a eu pour effet de modifier le sens du résultat. On voulait `abs( (x+y) - (x-y) )` et on a obtenu `abs( x+y - x-y )`. L'utilisation de macros avec arguments est très délicat et peut être source de nombreuses erreurs.

4.4 Pointeurs

Une variable entière est associée au moment de sa déclaration à une case mémoire. Elle permet d'accéder à la valeur stockée à cet emplacement ou bien de modifier cette valeur. Une fois que la variable est allouée dans la mémoire, la case mémoire qu'elle désigne ne change pas. Il peut être utile dans certaines situations de manipuler des variables que l'on peut associer librement au cours de l'exécution à plusieurs cases mémoires. C'est l'objet des pointeurs. Un pointeur sur un entier est une adresse qui désigne une case de la mémoire. Une variable de type pointeur sur un entier peut ainsi contenir l'adresse d'une case mémoire, comme d'autres variables, cette valeur peut être modifiée au cours de l'exécution ce qui permet de désigner au cours de l'exécution plusieurs cases mémoires.

4.4.1 Manipulations de pointeurs

Le pointeur est une adresse. Une adresse correspond à une case mémoire mais ces deux notions sont différenciées. Il y a des opérations explicites pour passer de l'une à l'autre des notions.

Opération `*`

Si p est un pointeur sur un type t, sa valeur est une adresse à partir de laquelle est stockée un objet de type t. On désigne par $*p$ la case mémoire associée à cette adresse. Cette case mémoire peut s'utiliser exactement comme une variable de type t, on peut accéder ou modifier la valeur. C'est donc un cas particulier de valeur gauche.

<val-gauche> `::=` `"*"` *<expr>*

Opération `&`

Inversement, si on a une variable x (ou plus généralement une valeur gauche), on peut calculer son adresse (à condition que la variable ne soit pas stockée dans un registre ...). On désigne par `&` x, l'expression dont la valeur est l'adresse de x.

<expr> `::=` `"&"` *<val-gauche>*

Les opérations `*` et `&` sont inverses l'une de l'autre. Ainsi, `*(&` x`)` et x désignent la même case mémoire et `&(*`x`)` et x représentent la même adresse.

Déclarations de pointeurs

La déclaration d'une variable x de pointeurs sur un entier, se fait par la commande :

```
int *x;
```

Une manière de lire cette déclaration est de voir **int** `*` comme le type des pointeurs sur des entiers. D'ailleurs, un prototype de fonction f dont l'argument est un pointeur sur un entier s'écrit.

```
int f (int *)
```

Cependant cette lecture est trompeuse. En effet si on déclare

```
int *x,y;
```

alors cela ne déclare pas deux variables de type pointeur sur un entier, mais une variable x qui est un pointeur sur un entier et y qui est un entier. La bonne manière de lire les déclarations est que l'expression `*`x représente une case mémoire et joue donc le rôle d'une variable de type **int**.

L'introduction de pointeurs nous amène à généraliser la syntaxe des déclarations. Pour l'instant, nous avions comme déclarations possibles des variables x, des tableaux x`[`n`]` et maintenant des pointeurs `*`x mais on peut aussi faire des tableaux de pointeurs, des pointeurs de tableaux, des pointeurs de pointeurs ou des tableaux de tableaux. Cela nous amène à introduire la catégorie des déclarateurs *<declarateur>*. Une déclaration aura la fome syntaxique :

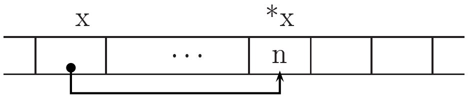
<declaration> `::=` *<type>* *<declarateur>*

Un déclarateur peut être une expression complexe combinant pointeur et tableau, il peut donc être nécessaire de parenthéser cette expression, ce qui nous amène à la syntaxe suivante des déclarateurs.

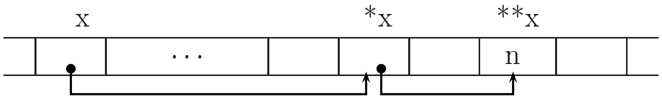
<declarateur> `::=` *<ident>*
<declarateur> `::=` *<declarateur>* `"["` *<constante>* `"]"`
<declarateur> `::=` `"*"` *<declarateur>*
<declarateur> `::=` `"("` *<declarateur>* `")"`

Les déclarations doivent se lire de la manière suivante :

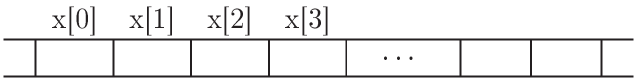
`int *x;` `x` est un pointeur sur un entier, c'est une variable dont la valeur est l'adresse d'une case mémoire pouvant contenir un entier.



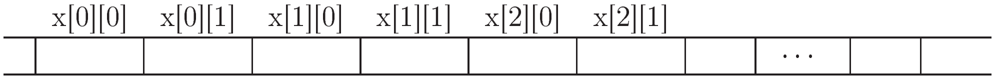
`int **x;` est un pointeur sur un pointeur sur un entier, c'est une variable dont la valeur est l'adresse d'une case mémoire pouvant contenir une adresse sur une case mémoire pouvant contenir un entier.



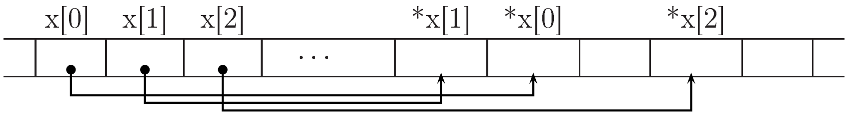
`int x[4];` est un tableau de taille 4 contenant des entiers, qui correspond à 4 cases mémoire allouées consécutivement dans la mémoire.



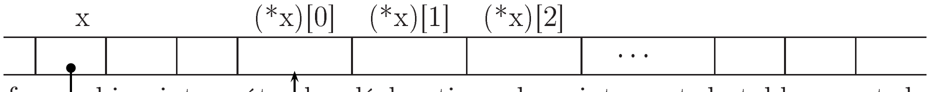
`int x[3][2];` est un tableau à deux dimensions, 3 lignes et 2 colonnes. Les éléments sont rangés par ligne.



`int *x[3];` doit se lire comme `int *(x[3]);` ce qui signifie que `x[i]` est un pointeur sur un entier et donc que `x` est un tableau de pointeurs sur des entiers.



`int (*x)[3];` signifie que `* x` est un tableau de 3 éléments entiers et donc que `x` est un pointeur sur un tableau de 3 éléments.



La clef pour bien interpréter les déclarations de pointeur et de tableau, est de se rappeler qu'il faut lire les déclarations de typage *à l'envers*. Au lieu de dire que x est un tableau de pointeur on dit que à partir d'un élément de x, on peut accéder via une indirection à un entier. L'autre point à mémoriser est que dans l'expression `*x[k]` c'est l'accès dans le tableau qui a la priorité, il faut donc lire `*(x[k])`.

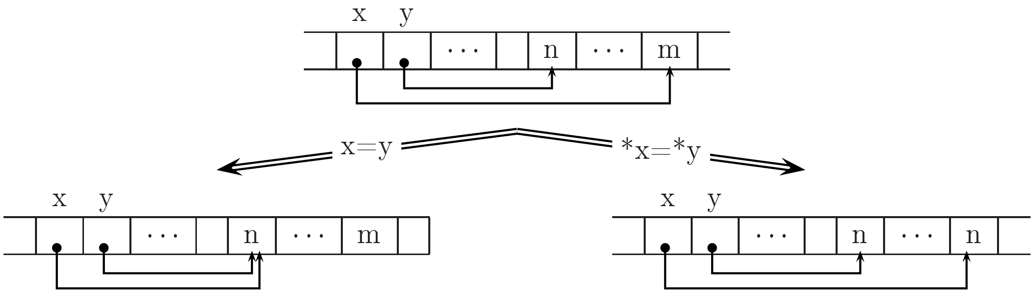
Pointeurs et affectation

Lorsque l'on déclare une variable x de type pointeur sur un entier, alors c'est une variable comme une autre donc elle correspond à une case mémoire dans laquelle est stockée une adresse correspondant à une autre case mémoire. On peut donc accéder à la valeur de x (une adresse) ou bien la modifier, mais on peut aussi accéder à la valeur de la case référencée par x (un entier) ou modifier cette valeur.

Si x et y sont deux variables de pointeur, alors il est possible de réaliser l'affectation x=y qui aura pour effet que x et y contiennent la même adresse sur une nouvelle case mémoire. Toute mise à jour de cette case mémoire aura pour effet de modifier les valeurs de $*x$ et $*y$. On peut également réaliser l'affectation $*x$=$*y$ qui consiste à mettre la valeur de la case référencée par

y dans la case référencée par x, cette opération ne modifie pas les valeurs de x et y qui restent independants.

Ce comportement est illustré sur la figure suivante :



Passage de paramètres

L'utilisation de pointeurs permet de programmer des fonctions dont le passage d'arguments sera analogue au passage par référence. Ainsi, si on veut faire la fonction d'échange, au lieu de prendre en argument deux variables entières x et y dont les valeurs doivent être échangées, on prend en argument deux pointeurs sur des entiers. La fonction a pour effet d'échanger les valeurs dans les cases mémoires associées à ces pointeurs. La fonction se programme de la manière suivante :

```
int echange(int *x, int *y) {int tmp = *y; *y = *x; *x = tmp; }

int main() {
  int x = 3, y = 5;
  echange(&x,&y);
  printf("x=%d, y=%d\n",x,y);
}
```

On note les changements par rapport à la tentative précédente de procédure d'échange en C. Les paramètres x et y de la fonction `echange` sont maintenant déclarés comme des pointeurs sur des entiers. Le corps de la fonction fait la suite d'affectations sur les cases mémoires correspondantes notées $*x$ et $*y$. Dans le corps de `main` on se donne deux variables ordinaires x et y de type entier. Pour appliquer la fonction `echange` il faut fournir comme arguments les adresses de ces variables ce qui se fait via les expression $\&x$ et $\&y$. On constate que l'échange a bien été réalisé.

Arithmétique de pointeurs

Les pointeurs sont des adresses mais le type des pointeurs n'est pas identifié au type des entiers.

Une seule exception, 0 est une valeur possible pour un pointeur. Cette adresse ne correspond à une aucune donnée valide. On utilise plus spécifiquement la constante `NULL` définie égale à 0 dans `<stdio.h>` pour désigner cette expression de pointeur particulière.

Certaines opérations *arithmétiques* peuvent être effectuées sur les expressions représentant des pointeurs. Un pointeur représente l'adresse d'une case mémoire. Dans un tableau par exemple, les données sont organisées dans des cases mémoires successives de même taille. L'opération d'ajout d'un entier n à un pointeur p renvoie un pointeur positionné sur une case mémoire située n cases après la case référencée par p. L'intérêt de cette opération est qu'elle tient compte de la taille de la case qui dépend du type associé au pointeur. Ainsi en augmentant de 1 le pointeur, on aura un pointeur sur la case suivante, ceci quelle que soit sa taille. En particulier, on peut utiliser les opérateurs d'incrémentation sur les pointeurs et écrire : `p++`, `++p`, `p--`, `--p`.

Si deux pointeurs p et q correspondent à des adresses sur le même tableau, on peut comparer leurs positions respectives par l'opération $p < q$. On peut aussi calculer l'entier (noté $q - p$) correspondant au décalage entre ces deux pointeurs.

4.4.2 Tableaux et pointeurs

On a vu comment déclarer des tableaux de taille fixe. Au moment de la déclaration, l'espace mémoire est réservé pour stocker le tableau, il faut donc connaître sa taille.

Initialisation Il est possible d'initialiser un tableau par un ensemble de valeurs. Ces valeurs sont écrites entre accolades `{` et `}` et séparés par des virgules. Dans le cas où l'initialisation est explicite on peut omettre l'indication de taille qui sera calculée par le compilateur. La déclaration :

```
int t [] = {1,2,3,4}
```

réserve la place pour un tableau de 4 éléments.

Un tableau vu comme pointeur En C, les pointeurs et les tableaux sont fortement liés. De fait si t est un tableau de taille n, alors $t[0], \ldots, t[n-1]$ sont des cases mémoires. Le nom t lui-même est une expression qui représente l'adresse de la première case du tableau, c'est-à-dire `&`$t[0]$. En particulier $t[i]$ la i-ème case mémoire a pour adresse `&`$t[i]$ qui est la même chose que `&`$t[0] + i$, c'est-à-dire $t + i$. Le compilateur C traduit effectivement l'accès dans un tableau $t[i]$ par l'opération sur les pointeurs `*`$(t + i)$.

Cependant, le nom t n'est pas une variable, ce n'est pas une valeur gauche, il ne peut pas apparaître à gauche d'un signe d'affectation ni dans une incrémentation.

On peut écrire le programme suivant qui affecte un tableau à une variable de pointeur puis qui accède aux éléments :

```
main () {
  int u [] = {1,2,3};
  int * x;
  x = u;
  printf("u[1]=%d\n",u[1])
}
```

Par contre l'affectation $u = x$ n'est pas autorisée car u est associé à un emplacement fixe dans la mémoire.

La situation est différente lorsqu'un tableau apparaît comme paramètre d'une fonction. Dans ce cas il est traité comme un pointeur sur la première case du tableau.

Il y a alors équivalence entre avoir un argument x passé comme paramètre de type <*type*> x `[]` et un argument x passé comme paramètre de type <*type*> `*` x.

4.4.3 Chaînes de caractères

Une chaîne de caractères est représenté comme un tableau de caractères. Dans la représentation interne, le tableau se termine par un caractère spécial : le caractère de code ASCII 0, aussi noté \0. Cela permet aux programmes de repérer aisément la fin de la chaine, qui contiendra donc un élément de plus que de caractères dans la chaîne.

En prenant en compte la correspondance entre pointeur et tableau, on voit qu'une chaîne passée en argument sera de fait un pointeur sur un caractère.

Copie de chaîne

Soit s et t deux chaînes de caractères, supposons que l'on veuille recopier la chaîne t dans la chaîne s, en supposant que la taille de s soit supérieure à la taille de t.

On ne peut pas faire $s = t$ car s n'est pas une valeur gauche. Si s et t sont déclarés comme des pointeurs sur des caractères, alors l'affectation est possible mais son effet est de faire pointer s sur la chaîne associée à t, cela ne fait donc pas une recopie (en particulier un changement sur un caractère de t affecterait s).

Le programme suivant effectue la copie

```
int i = 0;
do { s[i] = t[i]; i++; } while (t[i] != '\0')
```

Exercice : dire pourquoi une boucle **while** simple ne fonctionne pas.

Une manière plus classique d'écrire cela en C est d'utiliser le fait que l'affectation `s[i] = t[i]` a pour valeur `t[i]`.

```
int i = 0;
while ((s[i] = t[i]) != '\0') i++;
```

Maintenant, si on considère s et t comme des pointeurs, on peut se passer de l'indice i et incrémenter directement les pointeurs. L'accès de tableau $t[i]$ sera remplacé par l'accès à un pointeur $*t$:

```
while ((*s = *t) != '\0') { t++; s++; }
```

Comme la valeur de $s++$ est la même que celle de s, on peut intégrer l'incrémentation dans le test et écrire simplement :

```
while ((*s++ = *t++) != '\0') ;
```

En utilisant le fait que '\0' est de fait la valeur zéro, on remarque que d'un point de vue test, `(*s++ = *t++) != '\0'` est équivalent à `(*s++ = *t++)`. La copie de la chaîne t dans la chaîne s se fait à l'aide de la simple boucle :

```
while (*s++ = *t++);
```

4.4.4 Tableaux de pointeurs

Lorsque l'on déclare un tableau de dimension 2, on doit réserver immédiatement la place mémoire correspondante. Cette forme carrée n'est pas toujours très adaptée. Ainsi si on déclare un tableau de chaine de caractères sous la forme

```
char t [] [8] = {"Ceci", "est", "un", "exemple"};
```

alors il faut réserver la place pour 4 × 8 caractères consécutifs puisque le mot le plus long `"exemple"` occupe en mémoire 8 caractères.

De plus, réordonner les éléments du tableau nécessite de les recopier.

Une alternative est de définir `t` explicitement comme un tableau de pointeur.

```
char * t [] = {"Ceci", "est", "un, "exemple"};
```

Dans ce cas, le tableau t lui-même ne contient que 4 cases comportant les adresses des chaînes qui chacune peuvent être allouées consécutivement sans avoir à leur donner à toutes le même format.

De plus on peut réordonner les éléments du tableau sans avoir à effectuer de copie, simplement en changeant les pointeurs de position.

4.4.5 Pointeurs et généricité

Beaucoup d'algorithmes fonctionnent indépendament des structures de données qu'ils manipulent. C'est le cas en particulier des algorithmes de recherche ou de tri, des tables d'associations…

Il est intéressant de pouvoir écrire un code qui soit largement indépendant des données qu'il utilise. Malheureusement, les données ne font pas toutes la même taille, le code à exécuter va donc être différent. Une manière de régler ce problème est de traiter les données via des pointeurs. En effet un pointeur fait toujours la même taille, qu'il pointe sur un entier, un caractère ou une chaîne de caractère. Malheureusement pour manipuler un pointeur on doit donner le type des données sur lequel il pointe. Pour contourner ce problème, le langage C autorise la manipulation du type des pointeurs vers **void**. Toute variable de pointeur sur un type t peut être castée vers le type **void** `*`. On peut ensuite la recaster vers son type d'origine. Cela permet d'oublier la structure. Cette possibilité est particulièrement utile en conjonction avec l'utilisation de pointeurs de fonctions qui permettent de passer des fonctions en argument à d'autres fonctions. Les bibliothèques contiennent de telles constructions pour des fonctions de recherche et de tri.

4.5 Les structures

Nous avons jusqu'à présent essentiellement manipulé comme type de base des types entiers. Or beaucoup de problèmes informatiques nécessitent de manipuler des données complexes. Ces données ont souvent de bonnes propriétés structurelles qui doivent se refléter dans leur représentation informatique.

Ainsi si on veut représenter un point avec deux coordonnées entières x et y, on peut d'un point de vue mathématique le représenter par l'entier 2^x3^y, seulement l'accès aux composantes du point qui est une opération essentielle ne sera pas immédiate du point de vue calculatoire, de plus on atteindra rapidement des problèmes de dépassement.

Il faut donc trouver des représentations efficaces tant sur la place occupée en mémoire que sur l'efficacité des opérations élémentaires sur les structures. Les points par exemple peuvent se représenter sous la forme d'un tableau de deux entiers. Mais on ne pourra pas utiliser la même technique si les deux composants ne sont pas de même nature. De plus, les tableaux qui sont assimilés en C à des pointeurs ne peuvent pas se manipuler comme des valeurs ordinaires dans les fonctions (nous reviendrons sur ce point dans la partie 4.5.5 traitant de la mémoire).

Dans ce chapitre, nous passons en revue un certain nombre de structures courantes et montrons comment les représenter dans le langage C.

4.5.1 Types énumérés

Beaucoup de problèmes manipulent des ensembles finis. Par exemple les mois de l'année, les couleurs de base ... Plutôt que d'utiliser des entiers, il est recommandé d'introduire des types énumérés. Ceux-ci se comportent comme des entiers mais peuvent être représentés de manière efficace sur un nombre réduit de bits. Par ailleurs, il n'est pas obligatoire de fixer les valeurs des constantes énumérées. Le compilateur leur fixe arbitrairement des valeurs consécutives. La déclaration d'un type énuméré se fait de la manière suivante :

<declaration> `::=` **enum** `<ident>` `"{"` *<ident>* { `","` *<ident>* } `"}"` `";"`

Par exemple, on pourra simuler un type des valeurs booléennes avec la déclaration :

```
enum bool {false,true};
```

À la suite de cette déclaration, on a une nouvelle expression de type **enum** `bool`.

<type> `::=` **enum** `<ident>`

Et deux valeurs `true` et `false` de type **enum** `bool` (qui valent respectivement 0 et 1). Ce type **enum** `bool` est un type de petit entier, les objets de ce type seront promus au rang d'objets de type **int** lorsqu'on leur appliquera des opérations entières.

On peut donc par exemple, définir une fonction pour le ou logique :

```
enum bool or (enum bool x, enum bool y) { return (x ? true : y); }
```

Les compilateurs ne vérifient pas que l'affectation à une valeur dans un type énuméré se fait bien avec une valeur correcte. Ainsi la définition suivante sera aussi bien acceptée.

```
enum bool or (enum bool x, enum bool y) { return 4; }
```

Cependant l'utilisation du type énuméré améliore la lisibilité du programme en explicitant les intentions du programmeur.

Par défaut, les valeurs des constantes énumérées sont des entiers successifs en partant de zéro, mais on peut aussi fixer certaines constantes, les suivantes seront numérotées de manière successive. Ainsi la déclaration suivante introduit les mois de l'année en les numérotant de 1 à 12.

```
enum month { JAN=1,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC };
```

On peut aussi fixé arbitrairement les valeurs des constantes. Le type suivant définit un type énuméré pour les caractères d'espacement :

```
enum space { TAB='\t', SPA=' ', NL='\n', CR='\r', FF='\f' };
```

Deux constantes d'un type énuméré peuvent avoir la même valeur.

4.5.2 Définitions de structures

De nombreux programmes ont besoin de manipuler des données comportant plusieurs composantes. Par exemple un répertoire téléphonique élémentaire doit comporter un ensemble d'enregistrements comportant un nom, un numéro de téléphone. Les programmes un peu archaiques (on retrouve ce genre de pratique dans des programmes de bas niveau) stockent ces données dans des tableaux avec certaines conventions sur la place occupée par chaque partie. Les langages de programmation modernes offrent des primitives de langage qui permettent d'effectuer les opérations de mise à jour et d'accès aux composants.

On appelle *enregistrement*, (ou *structure*) les types de données composées. L'idée est d'attacher des étiquettes (des noms) à chacun des composants intéressants. On pourra ainsi accéder aux différentes composantes en utilisant les étiquettes. La spécification d'une structure se fait en donnant les étiquettes et pour chacune d'elle le type de la composante associée.

<*type*> `::=` **struct** `"{"` { <*type*> <*declarateur*> `";"` } `"}"`

Pour faciliter l'écriture, on peut introduire une déclaration de structure :

<*declaration*> `::=` **struct** `<ident>` `"{"` { <*type*> <*declarateur*> `";"` } `"}"` `";"`

On aura alors une nouvelle construction de type :

<*type*> `::=` **struct** `<ident>`

L'expression **struct** `<ident>` est une abréviation pour le type **struct** { ...} correspondant.

Par exemple l'entrée du répertoire téléphonique peut se déclarer de la manière suivante :

```
struct repertoire {char * nom; int tel;};
```

Ce type a deux *champs*, le premier d'étiquette *nom* contient une chaîne de caractères (un pointeur sur un caractère) le second contient un entier.

D'un point de vue opérationnel, une variable de type une structure est un ensemble de cases mémoires consécutives comportant les différents champs.

On peut initialiser une structure en donnant les valeurs des différents champs, par exemple :

```
struct repertoire me = { "Paulin", 56635 };
```

Les différents champs d'une structure peuvent être référencés en utilisant les étiquettes. Cela se fait sous la forme <*expr*>.<*ident*>. Cette expression correspond à une case mémoire qui peut être modifiée ou lue. On a donc une nouvelle construction pour les valeurs gauches et les expressions.

<*val-gauche*> `::=` <*val-gauche*>`"."`<*ident*>
<*expr*> `::=` <*expr*>`"."`<*ident*>

Par exemple on peut changer le numéro de téléphone dans notre répertoire en faisant :

```
me.tel = 56536;
```

Les structures ressemblent à des tableaux, cependant les éléments qui la composent n'ont pas à être de même nature et l'accès se fait de manière symbolique plutôt que par un indice entier. Les opérations sur les structures sont également plus permissives. On peut affecter une structure ou la renvoyer comme résultat d'une fonction.

Contrairement aux tableaux qui sont des pointeurs, les structures sont des valeurs au même titre que les entiers ou les flottants. Une variable de type structure représente une case mémoire assez grande pour contenir l'ensemble de la structure. Il est possible d'affecter globalement une structure. Ainsi, lors d'une affectation entre deux structures `s1 = s2`, il y a une recopie des champs de `s2` vers les champs de la structure `s1`.

Structures et fonctions

Les structures sont manipulées par valeur également lors de l'appel de fonctions. Ainsi lorsque l'on passe une structure en argument à une fonction, les composantes de la structure sont recopiées sur la pile. Ainsi, on peut définir une fonction de changement d'un numéro de téléphone dans une entrée du répertoire :

```
void changetel (struct repertoire rep, int t) { rep.tel = t; }
```

Mais on peut constater que l'application de cette fonction à l'entrée `me` ne modifie pas la valeur de `me`. En effet, cette fonction ne fait que modifier l'une des cases mémoire associées au paramètre `rep` qui sont créées au moment de l'appel et détruit ensuite.

Si on simule le même exemple avec un tableau à deux éléments entier :

```
void changetab (int rep[], int t) { rep[1] = t; }
```

Alors on constate que le premier champ d'un tableau passé en argument de cette fonction est modifié. En effet, au moment de l'appel, seul le pointeur sur le tableau est passé en argument et les modifications sont faites sur le tableau.

Structures comme résultat Les structures peuvent être renvoyées comme résultat d'une fonction. Ainsi la fonction suivante crée une nouvelle structure initialisée par les arguments de la fonction.

```
struct repertoire mk_entry (char * nom, int tel) {
  struct repertoire tmp = { nom, tel };
  return tmp;
}
```

Au moment de l'appel de fonction, la place pour la valeur de retour est réservée. On alloue ensuite la place pour les arguments `nom` et `tel` ainsi que pour la variable locale `tmp`. L'instruction **return** `tmp` a pour effet de recopier la valeur de `tmp` (la structure) à l'emplacement réservé pour le retour de la fonction. Lorsque l'on sort de la fonction, l'espace utilisé pour les arguments et les variables locales est libéré mais la valeur de retour reste accessible.

Le même mécanisme ne fonctionne pas pour un tableau. Si on compile la fonction suivante :

```
int * mk_tab (int x1, int x2) {
 int tmp [2] = { x1, x2 };
 return tmp;
}
```

On obtient un avertissement :

```
warning: function returns address of local variable
```

En effet, la fonction renvoie la valeur de `tmp` qui est un pointeur sur la première case mémoire allouée pour le tableau `tmp`. Mais cette partie de la mémoire est libérée après l'exécution de la fonction, on a donc comme valeur un pointeur *dans le vide*, ce qui n'est pas recommandé…

4.5.3 Composer les structures

Les structures sont des types comme les autres. On peut donc les utiliser librement. Par exemple, une structure peut contenir un champ dont le type est une autre structure. On peut former des tableaux de structures ou des pointeurs sur des structures.

L'accès à une composante de ces objets se fait en composant les différentes opérations d'accès. Ainsi, on peut introduire un tableau formé d'entrées de répertoire.

```
#define MAXREP 100
struct repertoire rep [100];
int free = 0;
```

L'ajout de la structure `me` pourra se faire en affectant la structure complète :

```
rep[free++] = me;
```

Si on veut accéder au nom de la dernière entrée, on accède au champ de la structure :

```
printf("%s\n", rep[free-1].nom);
```

On peut aussi définir un pointeur sur une structure.

```
struct repertoire * prep;
```

Pour accéder aux composants, on utilise la notation `(* prep).nom` et `(* prep).tel`. Attention, les parenthèses sont importantes, en effet `* prep.nom` est parenthésé par défaut comme `* (prep.nom)` ce qui est incorrect car `prep` n'est pas une structure.

Notation *p* `->` *etiquette*

Les pointeurs sur les structures sont très utilisés car ils permettent de manipuler les structures sans les recopier systématiquement. Aussi la notation *p* `->` *etiquette* est une abréviation pour l'expression `(*` *p*`).`*etiquette*. Elle est licite si *p* est un pointeur vers une structure qui a un champ nommé *etiquette*. Elle représente une valeur gauche si *p* est une valeur gauche.

Précédences La notation `->` a la même précédence que les opérateurs `[]` et `.` et associe à gauche. La précédence de ces opérateurs est plus importante que celle de l'opération `*` de déréferenciation ou que les opérateurs d'incrémentation ou de décrémentation.

L'exemple suivant crée artificiellement des structures imbriquées et illustre les questions de précédence :

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

struct rect {
  struct point * bas;
  int haut [2];
};

struct rect fig [1];
struct rect * prect;
```

Le type `point` est une structure simple composée de deux entiers. Le type `rect` est une structure dont le champ `bas` est un pointeur sur un point et le champ `haut` est un tableau de deux entiers. On introduit ensuite une variable `fig` qui est un tableau d'objets de type `rect` et une variable `prect` qui est un pointeur sur un objet de type `rect`. On indique dans la table suivante différentes expressions donnant accès aux différentes composantes de ces objets en indiquant le parenthésage implicite et des formes équivalentes.

`* fig[0].bas`	`* (fig[0].bas)`	
`fig[0].bas->y`	`(fig[0].bas)->y`	`(* (fig[0].bas)).y`
`prect->bas->x`	`(prect->bas)->x`	`(* (* prect).bas ).x`
`fig[0].haut[1]`	`(fig[0].haut)[1]`	
`prect->haut[1]`	`(prect->haut)[1]`	`((* prect).haut)[1]`

4.5.4 Types unions

Les structures permettent de représenter des types de données ayant plusieurs composantes. Une entrée du répertoire est composée d'une chaîne de caractère *et* d'un entier. Il est souvent utile de manipuler des données pouvant avoir des représentations physiques différentes. Par exemple, on peut représenter un type des valeurs numériques qui pourrait être un entier *ou* un réel.

Lorsque les données d'un type peuvent avoir plusieurs formes, une possibilité est de mettre les différentes formes possibles à plat dans une structure.

```
struct num { int ival; double dval; }
```

et de n'utiliser que la composante qui nous intéresse. Cependant chaque objet de type `num` occupera la place d'un entier plus un flottant alors qu'on n'a besoin que de l'un des deux. Pour produire une valeur dans le type `num` à partir d'un entier, il faut quand même produire une initialisation pour la partie flottante ce qui n'est pas plaisant. Les types **union** de C apportent une solution à ce problème. La syntaxe est la même que pour les structures :

<declaration> `::=` **union** `<ident>` `"{"` { *<type>* *<declarateur>* `";"` } `"}"` `";"`
<type> `::=` **union** `"{"` { *<type>* *<declarateur>* `";"` } `"}"`
<type> `::=` **union** `<ident>`

Le comportement est le même que pour les structures, sauf qu'au lieu de mettre les champs cote-à-cote, on les superpose. La taille d'un objet dans le type union est donc la taille nécessaire à la représentation du champ le plus grand.

En général, lorsqu'on manipule un objet dans un type union, on a besoin de savoir de quelle nature il est. Pour cela on ajoute une information dans un type énuméré qui reflète cette information. La structure finale pour représenter `num` pourrait donc être :

```
enum sort {INT,FLOAT};
struct num {
  enum sort s;
  union { int ival; double dval; } val
  };
```

Le type `num` est donc une structure dont le premier champ vaut 0 si on représente un entier et 1 si on représente un flottant. Le deuxième champ `val` contient un objet dans un type **union** qui peut être un entier ou bien un flottant. Si x est une variable de type `union` alors x`.val` est un objet de type union, x`.val.ival` est un objet de type **int**, et x`.val.dval` est un objet de type **double**.

Remarque: L'initialisation d'une union ne peut se faire que pour la valeur du premier champ. On pourra donc initialiser une variable :

```
struct num n1 = {INT, 4}
```

Ce qui aura pour effet de mettre 0 dans le champ `n1.s` et 4 dans `n1.val.ival`. Mais si on définit

```
struct num n2 = {FLOAT, 4.4}
```

Alors, cela aura pour effet de mettre 1 dans le champ `n2.s`, mais comme seul le premier champ de l'union est initialisé, cette instruction met aussi 4 dans `n2.val.ival` et le champ `n2.val.dval` reste non initialisé... Cet exemple montre que c'est de la responsabilité de l'utilisateur de garantir la cohérence entre le champ sur la sorte de l'objet et la nature de la valeur mise dans le champ `val`.

Pour manipuler les objets dans le type `num`, il est recommandé d'introduire et d'utiliser des fonctions de construction et d'accès.

```
struct num mk_int (int n) {
  struct num tmp;
  tmp.s = INT; tmp.val.ival=n;
  return tmp;
}
struct num mk_double (double f) {
  struct num tmp;
  tmp.s = FLOAT; tmp.val.dval=f;
  return tmp;
}
int est_double (struct num n) {
  return n.s;
}
int int_val (struct num n) {
  if (n.s) printf("Erreur"); else return n.val.ival;
}
int double_val (struct num n) {
  if (n.s) return n.val.dval; else printf ("Erreur");
}
```

La manipulation des objets de type `num` uniquement via ces fonctions garantit la cohérence de la représentation.

4.5.5 Gestion de la mémoire

Les langages de programmation utilisent pour stocker les données du programme un morceau de mémoire organisé sous forme de pile (*stack* en anglais). Cette mémoire est utilisée pour stocker les variables globales du programme, mais aussi pour stocker des valeurs intermédiaires dans le cas de calculs complexes ou encore pour les arguments et variables temporaires lors des appels de fonctions.

On a déjà vu qu'une fonction pouvait renvoyer une structure comme valeur, l'espace mémoire pour celle-ci est alors réservé sur la pile. On a vu également qu'il pouvait être préférable de manipuler plutôt des pointeurs sur des structures. Dans ce cas une fonction de création d'un nouvel objet doit renvoyer comme valeur une adresse, et cette adresse doit correspondre à une portion de la mémoire qui doit rester valide tant que l'on manipule l'objet.

La pluspart des langages utilisent pour ce propos une partie de la mémoire distincte de la pile et appelée le *tas* (*heap* en anglais). Cette partie de la mémoire permet de stocker des objets qui sont créés lors de certains appels de fonction et qui doivent survivre après la fin de l'appel.

Dans un langage comme C, l'allocation de mémoire dans le tas est explicite tout comme la libération de la mémoire. Dans un langage comme Java, l'allocation se fait de manière implicite lors de la création d'objets, il n'y a pas d'instruction explicite de libération de la mémoire, l'environnement d'exécution intègre une procédure spéciale de récupération de mémoire appelé GC pour *Glaneur de Cellules* ou *Garbage Collector* aussi appelé ramasse-miettes. Utiliser correctement la mémoire est complexe. Il y a deux problèmes possibles : le premier est de libérer de la mémoire dont certaines parties sont encore accessibles à partir des variables du programme, ce qui peut amener à des résultats faux; le second est de garder en mémoire des objets inutilisables, ce qui est correct mais inefficace.

Allocation de mémoire

La fonction de base pour réserver de la mémoire est définie dans la bibliothèque `stdlib.h` et s'appelle `malloc`. Cette fonction prend en argument un entier qui représente la taille de la donnée à stocker. Elle renvoie un pointeur sur une zone mémoire de taille suffisante pour stocker la donnée. Dans la mesure où la même fonction peut être utilisée pour allouer un bloc pour un entier ou un flottant ou bien une structure composée, son type de retour est **void** `*`. Le pointeur obtenu comme valeur peut ensuite être casté vers le type de l'objet à allouer.

Afin de déterminer la taille nécessaire au stockage d'un objet, le langage C propose l'opérateur `sizeof`. Cet opérateur prend en argument un objet o (ou un type t) et renvoie un entier non-signé qui est le nombre d'octets nécessaires à stocker un objet du type de o (ou du type t).

On peut ainsi écrire la fonction qui crée un objet `point` et renvoie un pointeur sur cette structure.

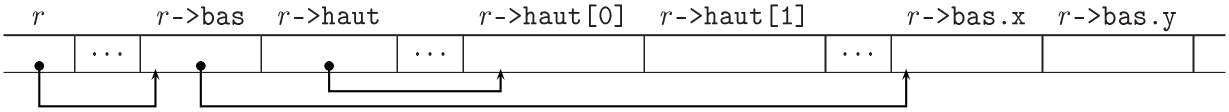
```
struct point * mk_point (int x, int y) {
  struct point * tmp;
  tmp = (struct point *) malloc (sizeof(struct point));
  tmp -> x = x; tmp -> y = y;
  return tmp;
}
```

On aurait pu remplacer l'appel `sizeof(struct point)` par `sizeof(* tmp)` qui a la même valeur.

La fonction `malloc` renvoie `NULL` si la mémoire n'a pu être allouée. Elle renvoie une portion de mémoire non initialisée. On peut de la même manière réserver de la place pour un tableau (la taille d'un tableau est le nombre d'éléments multiplié par la taille d'un élément) ou bien une chaîne de caractères (attention à ne pas oublier le `\0` final). Le pointeur obtenu sera casté vers un pointeur vers les éléments du tableau. Une fonction spéciale `calloc` prend en argument deux entiers n et s et renvoie un pointeur sur un bloc de mémoire permettant de stocker n éléments d'un type de taille s. Dans le cas de `calloc` le bloc alloué est initialisé à zéro.

Libération de la mémoire

La fonction `free` de la bibliothèque `stdlib.h` prend en argument un pointeur sur **void** et libère la partie de mémoire placée à cette adresse. Il faut faire attention à ne libérer que des parties de mémoire qui ont été allouées explicitement par `malloc` (ou `calloc`) et qui ne seront plus accédées par le programme. Il est possible de libérer la mémoire dans un ordre quelconque, cependant dans le cas de libération de structures comportant plusieurs pointeurs, il faut faire attention à conserver un accès valide aux sous-structures incluses afin de pouvoir également les libérer. En effet, si on reprend notre exemple des points, on peut se retrouver dans l'état où une variable du programme *r* est un pointeur vers un rectangle qui lui même est composé d'un pointeur sur un point et d'un tableau de deux entiers. L'organisation de la mémoire est alors :



Si on souhaite libérer la structure pointée par *r*, on risque de perdre les adresses pointées par *r*`.bas` et *r*`.haut`, il conviendra donc de libérer ces adresses en premier, ou bien de les préserver dans des variables temporaires.

4.5.6 Structures de données récursives

Une structure de données `t` est récursive si un de ses champs est déclaré avec un type faisant référence à `t` (on parle de type auto-référentiel). La structure récursive la plus simple est celle de liste chaînée. Chaque nœud est composé d'une valeur et d'un pointeur sur une structure du même type. On est obligé de faire l'auto-référence via un pointeur. En effet si le champ avait directement pour type la même structure, on ne serait pas en mesure de déterminer la taille d'un élément de cette structure, ni d'ailleurs de définir des objets finis de ce type.

On peut déclarer le type des listes chainées de la manière suivante.

```
struct noeud {
  int val;
  struct noeud * suiv;
};
```

On peut définir alors une fonction qui crée un pointeur sur un nœud avec une certaine valeur entière, le pointeur suivant étant mis à `NULL`.

```
struct noeud * mk_noeud (int x) {
  struct noeud * tmp = (struct noeud *) malloc(sizeof(struct noeud));
  tmp->val = x; tmp->suiv = NULL;
  return tmp;
}
```

Une autre manière de créer des listes chainées est d'ajouter un élément en tête de la liste :

```
struct noeud * add_noeud (int x, struct noeud * l) {
  struct noeud * tmp = mk_noeud (x);
  tmp->suiv = l;
  return tmp;
}
```

On peut également imprimer les éléments de la liste en parcourant les éléments de la liste jusqu'à trouver le pointeur nul.

```
void print_noeud (struct noeud * l) {
  while (l!=NULL) { printf("%d\n", l->val); l=l->suiv; }
}
```

On pourra alors construire une liste en y ajoutant des éléments successifs. Il suffit d'introduire une variable qui est un pointeur sur un nœud et d'ajouter les éléments un par un :

```
  struct noeud *tmp = NULL;
  tmp = add_noeud(1,tmp);
  tmp = add_noeud(2,tmp);
  tmp = add_noeud(3,tmp);
```

Attention, l'utilisation de pointeurs permet de créer des listes non linéaires. On crée par exemple aisément une liste bouclée. Pour cela il suffit d'utiliser une variable auxiliaire qui va venir se placer sur le dernier nœud de la liste puis de faire pointer le suivant sur la tête de la liste.

```
void boucle (struct noeud * l) {
  struct noeud * tmp = l;
  while (tmp->suiv != NULL) tmp = tmp->suiv;
  tmp->suiv = l;
}
```

Attention de telles listes font boucler la fonction d'impression. Pour résoudre ce problème, il faut mémoriser les nœuds visités.

On peut écrire une fonction qui libère les nœuds d'une liste qui ne boucle pas sur elle-même.

```
void free_noeud  (struct noeud * l) {
  struct noeud * tmp = l;
  while (l != NULL) {tmp = l->suiv; free(l); l = tmp;}
}
```

4.5.7 Définitions de type

Il est possible en C de définir des abréviations pour des types que ces types soient simples ou complexes. La forme de la syntaxe d'une définition de type est analogue à la forme de définition d'une variable. On précède la déclaration par le mot-clé **typedef** et on remplace le nom de la variable par le nom que l'on veut donner au type d'une variable qui se trouverait en même position dans la déclaration. Dans la suite on convient d'écrire les noms de type avec une majuscule. Quelques exemples de déclarations de types :

typedef int `Longueur`	`Longueur` est un synonyme de `int`
typedef char * `Chaine`	`Chaine` est un synonyme du type des chaînes de caractères (pointeurs sur un caractère)
typedef struct `point` { int x; int y; } `Point`	`Point` est un synonyme de la structure de point
typedef struct `point` * `PPoint`	`PPoint` est un synonyme du type des pointeurs sur un point

Les abréviations ainsi introduites peuvent être utilisées partout à la place du type correspondant.Il est fortement recommandé d'utiliser des définitions de type afin d'améliorer l'indépendance des développements par rapport à la représentation des données.

4.5.8 Exemple : l'allocation de mémoire

Nous allons développer un exemple montrant comment gérer la mémoire dans le tas en proposant une implantation des fonctions `malloc` et `free`.

Principe de l'algorithme On suppose que le tas est organisé en blocs de mémoire libres qui sont chaînés entre eux dans l'ordre croissant des adresses. Chaque bloc comporte une entête qui contient la taille du bloc ainsi qu'un pointeur sur le bloc libre suivant. Le dernier bloc pointe sur le premier. À la suite de l'entête se trouvent des cases de mémoire libres. Pour respecter les contraintes d'alignement, la taille du bloc est un multiple de la taille de l'entête. On choisit de conserver dans l'entête le rapport entre la taille complète du bloc (entête incluse) et la taille de l'entête. Lorsqu'on demande un espace mémoire, le système va chercher un bloc libre assez grand pour le contenir. Si le bloc obtenu est plus grand, il va le couper en deux créant un nouveau bloc de la taille souhaitée. Le pointeur renvoyé suit donc toujours une entête.

Structures de données

```
struct entete {struct entete *ptr; unsigned nbunit;};
typedef struct entete Entete;
```

Variables globales Une variable statique `base` contient l'entête initiale. Une variable statique `plibre` pointe sur l'entête du dernier bloc qui a été trouvé (initialement NULL). Cela permet que plusieurs appels successifs de `malloc` ne redémarre pas toujours du même endroit

```
static Entete base;
static Entete *plibre=NULL;
```

Déclarations La fonction `malloc` a pour prototype :

```
void *malloc(unsigned noctets)
```

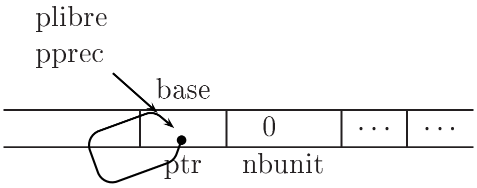
Elle utilise une fonction auxiliaire `plusmem` qui demande un bloc de mémoire au système. Cette fonction prend en argument le nombre d'unités que l'on souhaite obtenir, mais en général elle rendra un bloc plus grand qui sera ensuite découpé ceci pour limiter le nombre d'accès système.

```
Entete *plusmem(unsigned nu);
```

Dans le cours du programme, on manipule deux pointeurs sur des entêtes `pprec` et `p` qui vont varier, `p` va parcourir les cases libres jusqu'à trouver un bloc assez grand. Le pointeur `pprec` suit le pointeur `p` (il est sur l'entête précédent) de façon à pouvoir raccrocher les morceaux. Le nombre d'unités à stocker sera sauvegardé dans une variable `nu`.

```
Entete *p,*pprec;
unsigned nu;
```

Initialisation On place `pprec` sur le dernier bloc alloué, dans le cas ou celui-ci est NULL, on forme un tas trivial avec 0 cases libres.

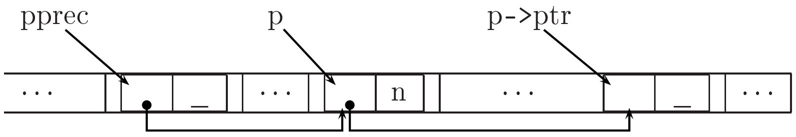


On compte ensuite le nombre d'unités à allouer. La fonction `malloc` prend en argument `nboctets` le nombre d'octets libres à retourner. Il faut placer de plus une entête et il faut trouver le plus petit nombre k, tel que $\mathbf{sizeof}(\texttt{Entete}) + \texttt{nboctets} \leq k \times \mathbf{sizeof}(\texttt{Entete})$ On se convainc que ce nombre est donné par la formule : $((\mathbf{sizeof}(\texttt{Entete}) + \texttt{nboctets} - 1)/\mathbf{sizeof}(\texttt{Entete})) + 1$.
L'initialisation correspond donc au code :

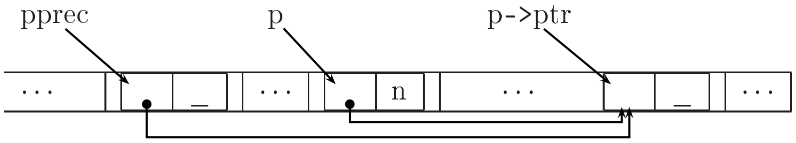
```
nu = ((sizeof(Entete) + nboctets-1)/sizeof(Entete))+1;
if ((pprec=plibre)==NULL)
  {pprec=&base; plibre=pprec;base.ptr=plibre; base.nbunit=0;}
```

On fait ensuite se déplacer les pointeurs `p` et `prec` le long des blocs libres en partant du suivant de `plibre` (le dernier bloc alloué) et en laissant `prec` un bloc derrière `p`. Si `p` se trouve sur un bloc de taille suffisante alors on l'utilise. Il y a deux cas, si le bloc a juste la bonne taille, il suffit de rebrancher le pointeur du précédent sur le suivant de `p` et de renvoyer le pointeur $\texttt{p}+1$ (adresse suivant l'entête). Si le bloc est trop grand, alors il faut le couper, on renvoie un morceau de mémoire en fin de bloc, en ayant pris soin de créer une nouvelle entête. Si on ne trouve pas de bloc de taille suffisante alors comme les blocs libres sont chainés en boucle, le pointeur `p` va revenir jusqu'à `plibre`. Il faudra alors demander de l'espace au système via la fonction `plusmem` que nous expliquons ensuite. Si cette fonction échoue, alors cela provoque un échec de la fonction.

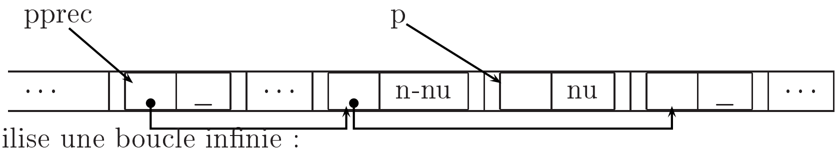
Les différentes situations sont illustrées par les figures suivantes :



Cas `nbunit` = `nu`



Cas `nbunit` > `nu`



La fonction utilise une boucle infinie :

```
for (p=pprec->ptr; ; pprec=p, p=p->ptr) {
    if (p->taille >= nu) {
      if (p->taille == nu) { pprec->ptr=p->ptr; }
      else {
          p->taille-=nu;
          p+=p->taille;
          p->taille=nu;
      }
     plibre = pprec;
     return (void *) p+1;
     }
     if (p == plibre)
       if ((p = plusmem(nu)) == NULL) return NULL;
}
```

La fonction `plusmem` utilise des fonctions systèmes de plus bas niveau pour obtenir un morceau de mémoire, elle convertit ce morceau de mémoire en un pointeur sur une entête dont elle initialise la taille. Puis elle utilise la fonction `free` afin d'insérer ce bloc dans la chaîne de blocs libres.

La fonction `free` Pour libérer un bloc il faut trouver à quel endroit l'insérer (pour respecter la contrainte de mettre les blocs en ordre croissant d'adresses). La fonction `free` prend comme argument un pointeur qui est supposé suivre une entête.

```
void free (void *pa)
```

Quand elle trouve où insérer le bloc, elle gère le cas où ce bloc est adjacent à un autre bloc et elle fusionne alors les deux blocs. Cette fonction utilise deux pointeurs `pb` qui pointe sur l'entête associé à `pa` et `p` qui se déplace le long de la chaîne de blocs libres à partir de `plibre`. Une première boucle sert à repérer l'endroit où insérer le bloc :

```
pb=(Entete *)pa-1;
for (p=plibre; !(p < pb && pb < p->ptr); p=p->ptr)
    if (p >= p->ptr && (pb > p || pb < p->ptr)) break;
```

La boucle sort lorsque l'adresse `pb` est entre deux adresses de blocs libres, cependant il faut traiter à part le cas où cette adresse est plus petite que celle du bloc d'adresse la plus petite ou plus grande que celle du bloc d'adresse la plus garnde. Cela se produit au moment où le pointeur `p->ptr` pointe sur une adresse plus petite que `p`. L'instruction `break;` provoque alors la sortie de la boucle.

On commence par joindre le nouveau bloc à son suivant, c'est-à-dire l'actuel suivant de `p`. On regarde s'il faut fusionner ces deux blocs.

```
if (pb + pb->taille == p -> ptr)
   { pb -> taille += p -> ptr -> taille;
      pb -> ptr = p -> ptr -> ptr; }
else pb -> ptr = p -> ptr;
```

On raccroche ensuite le nouveau bloc par le haut au bloc pointé par `p`. Là-aussi, on regarde le cas où il faut fusionner deux blocs.

```
if (p + p->taille == pb)
   { p -> taille += pb -> taille;
      p -> ptr = pb -> ptr; }
else p -> ptr = pb;
```

On met ensuite à jour `plibre` sur `p` qui est le dernier pointeur rencontré avant le bloc libéré.

```
plibre = p;
```

Remarque Ce programme présente une version simplifiée de `malloc`. Dans le livre [3], on trouvera une version qui prend en compte les contraintes d'alignement de la machine.

4.6 Les entrées/sorties

Les entrées-sorties en C se font principalement par des fonctions des bibliothèques. Le comportement de ces fonctions est décrit dans la norme ANSI, ce qui rend les programmes largement portables.

4.6.1 Les arguments de `main`

Il y a plusieurs manières d'effectuer les entrées d'un programme écrit en C. L'une d'elles consiste à passer des arguments à l'exécutable.

On a vu que tout programme devait comporter une fonction `main`. L'exécutable créé à la compilation (par défaut `a.out`) lance l'exécution de la fonction `main`. Il est possible d'appeler l'exécutable avec des arguments, ceux-ci seront passés à la fonction `main`. La fonction `main` peut être déclarée avec deux arguments. Le premier argument est de type **int**. Traditionnellement appelé `argc`, il aura pour valeur le nombre d'arguments de la commande (en comptant le nom de l'exécutable lui-même). Le second argument est un tableau de chaîne de caractères qui comporte les arguments eux-mêmes. Le découpage de la ligne de commande se fait en coupant les mots suivants les caractères d'espacement. Le premier élément du tableau d'argument contient le nom de l'exécutable. Le tableau a `argc+1` éléments, le dernier élément (`argv[argc]`) étant un pointeur sur `NULL`. Il peut être nécessaire de convertir les arguments d'entrée qui sont des chaînes de caractères en des entiers ou flottants. Certains des arguments serviront en général à fixer les options du programme.

Dans la fonction `main`, la sortie se fait traditionnellement par la fonction `exit` plutôt que par l'opérateur **return**.

4.6.2 Fonctions élémentaires `getchar/putchar`

Les fonctions `getchar` et `putchar` sont définies dans la bibliothèque `<stdio.h>`. Elles ont les prototypes suivants :

```
int getchar(void)
int putchar(int)
```

La fonction `getchar` arrête l'exécution en attente d'un caractère sur l'entrée standard (en général le clavier). Il est possible de mettre les entrées attendues dans un fichier et de le rediriger via le shell.

```
program < fichiers_entrées
```

La fonction `putchar`(c) envoie le caractère c sur la sortie standard, renvoie le résultat c ou bien la valeur spéciale `EOF` (pour *End Of File* : fin de fichier) s'il y a eu un problème.

4.6.3 Commande d'impression `printf`

Nous avons déjà utilisé cette commande. Elle permet d'imprimer des données diverses sous forme de caractères en respectant un format d'impression. Elle renvoie un entier qui est le nombre de caractères imprimés. C'est une fonction particulière car elle prend en argument un nombre variables d'expressions. Le premier argument décrit le format d'impression et conditionne le nombre et le type des autres arguments.

Un *format* est une chaîne de caractères qui contient des caractères ordinaires ou des commandes de conversion. Une commande de conversion commence par le caractère % et se termine par le caractère de type `d` (conversion d'entier) ou `s` (conversion de chaîne) ou `f` (conversion de flottant) ou `c` (conversion de caractère) . Entre le % et le caractère final de type, on peut mettre dans l'ordre :

- le caractère `-` pour indiquer que l'affichage doit être aligné à gauche (le défaut est un alignement à droite)
- un nombre n donnant le nombre minimal de caractères sur lequel imprimer l'argument (en ajoutant si nécessaire des blancs)
- le caractère `.` suivi d'un nombre p (la précision) donnant dans le cas des flottants le nombre maximal de chiffres après la virgule; dans le cas des chaînes, le nombre maximal de caractères à écrire et dans le cas des entiers, le nombre minimal de chiffres à imprimer

Chacun de ses éléments peut ou non apparaître dans la conversion. Deux caractères %% consécutifs sont interprétés comme le caractère ordinaire %.

L'indication de largeur ou de précision peut être remplacée par le signe `*`, auquel cas la fonction `printf` prend en plus un argument entier qui précisera le nombre à l'exécution.

Même si le plus souvent, le format est une constante de chaîne, ce n'est pas une obligation et on peut mettre n'importe quelle chaîne. Ainsi si on a une chaîne de caractères s, alors la commande

```
printf(s);
```

imprime la chaîne s mais si celle-ci contient des commandes de conversion alors l'interprétation sera incorrecte. Pour imprimer une chaîne quelconque, il est plus sûr d'écrire :

```
printf("%s", s);
```

Cette possibilité a pour conséquence que la compatibilité entre le format indiqué et le nombre et le type des arguments n'est pas vérifiée à la compilation. Des incompatibilités provoqueront des erreurs d'exécution.

La fonction `sprintf` travaille de la même manière que `printf`. Cependant, au lieu d'imprimer la chaîne formatée sur la sortie standard, celle-ci est stockée dans une chaîne passée en premier argument et qui doit être assez grande pour stocker le résultat. Le prototype de la fonction `sprintf` est donc :

`int sprintf (char *result, char *format,` $\mathtt{arg}_1,\ldots,\mathtt{arg}_n$`)`

Exemples Nous illustrons le comportement des commandes de conversion sur quelques exemples. La variable `c` de type `string` a pour valeur `"Hello World"` et la variable `n` de type `int` a pour valeur `12345`.

Instruction	Affichage
`printf("Imprimer %s!\n",c)`	`Imprimer Hello World!`
`printf("Imprimer \"%s\"!\n",c)`	`Imprimer "Hello World"!`
`printf("\"%s\" au moins %d fois!\nl",c,n)`	`"Hello World" au moins 12345 fois!`
`printf("Imprimer %15s!\n",c)`	`Imprimer     Hello World!`
`printf("Imprimer %-15s!\n",c)`	`Imprimer Hello World    !`
`printf("Imprimer %15.5s!\n",c)`	`Imprimer           Hello!`
`printf("Imprimer %-15.5s!\n",c)`	`Imprimer Hello          !`
`printf("Imprimer %10d!\n",n)`	`Imprimer      12345!`
`printf("Imprimer %-10d!\n",n)`	`Imprimer 12345     !`
`printf("Imprimer %2d!\n",n)`	`Imprimer 12345!`
`printf("Imprimer %-2d!\n",n)`	`Imprimer 12345!`

4.6.4 Commande de lecture `scanf`

La fonction `scanf` permet de lire des entrées non pas caractère par caractère mais en lisant des suites de caractères et en prenant en compte des formats analogues à ceux de `printf`.

L'entrée est décomposée en mots qui sont des suites de caractères séparés par des caractères d'espacement (espace, tabulation , fin de ligne, retour chariot, tabulation verticale et saut de page).

Les formats sont composés de caractères ordinaires (qui doivent se retrouver sur l'entrée) ou des caractères de conversion qui débutent par le signe % et se terminent par le caractère de conversion. Les formats peuvent contenir un signe `*` qui indique de ne pas stocker la donnée correspondante et peut également contenir une valeur entière qui donne la largeur maximale de l'entrée à interpréter. Les caractères de conversion pour les entiers peuvent être modulés pour lire des entiers courts, longs ou non-signés.

Les arguments suivant le format de `scanf` doivent être obligatoirement des pointeurs sur des morceaux de mémoire capables de contenir les données. Ainsi pour lire un entier sur l'entrée standard et le stocker dans une variable i, on écrira :

```
int i;
scanf("%d",&i);
```

Pour lire une chaîne de caractères, on pourra écrire :

```
char chaine[20];
scanf("%s",chaine);
```

Dans le second cas, le tableau est directement un pointeur. On peut aussi utiliser des formats qui marient caractères de conversion et caractères standards. Ainsi pour lire un couple d'entiers sous la forme (n,m), on pourra utiliser les commandes suivantes :

```
int n,m;
scanf("(%d,%d)",&n,&m);
```

Le prochain appel de `scanf` ou d'une autre fonction d'entrée reprend la lecture au prochain caractère non lu. Ainsi on peut lire une série de points stockés dans un fichier et les ranger dans un tableau de la manière suivante :

```
#define MAX 10
typedef struct int abs; int ord; Point;
Point t[MAX];
main() {
  int i=0;
  while (i < MAX && (scanf("(%d,%d)\n",&t[i].abs,&t[i].ord)==2)) i++;
  for (i-- ; i >= 0; i--) printf("(%d,%d)\n",t[i].abs,t[i].ord);
}
```

Pour décomposer un texte en mots, on pourra utiliser le programme suivant :

```
main() {
  char mot[MAX];
  while (scanf("%s",mot)!=EOF)  printf("%s\n",mot);
}
```

`mot` en tant que tableau est déjà un pointeur. Il faut par contre que sa taille soit suffisante pour lire chaque entrée.

Remarque Attention à la gestion des blancs. Un champs est défini comme une chaîne de caractères différents des caractères d'espacement; il s'étend soit jusqu'au caractère d'espacement suivant, soit jusqu'à ce que la largeur du champs soit atteinte si elle est précisée.

4.6.5 L'accès aux fichiers

Il est parfois utile d'effectuer les entrées et sorties non pas via l'entrée et la sortie standard du programme (clavier, écran ou redirection du shell) mais via des fichiers.

La manière dont le fichier est manipulé sur la machine dépend du système d'exploitation. Les langages de programmation ont des fichiers une certaine représentation "logique", comme un flux d'entrée: ils utilisent un tampon, tableau de caractères qui sert à stocker une portion du fichier lu ou à écrire (afin de limiter les opérations d'accès aux fichiers qui peuvent être longues) ainsi qu'un indicateur sur la position de lecture dans le fichier; ils gardent également trace de possibles erreurs, de l'arrivée en fin de fichier, du droit de lire ou d'écrire . . .Le format d'échange avec les fichiers n'a pas besoin d'être connu par le programmeur.

En C, la représentation logique d'un fichier est un objet qui est un pointeur sur le type `FILE` défini dans la bibliothèque `<stdio.h>`

Dans cette bibliothèque sont prédéfinis des représentations logiques de l'entrée standard `stdin` (en général le clavier), de la sortie standard `stdout` (en général l'écran) et de la sortie en erreur `stderr` (on peut dans le shell rediriger différemment la sortie d'erreur et la sortie standard).

Lorsque l'on effectue une redirection au niveau du shell :

```
prog < fichier
```

Cela revient à rediriger l'entrée standard du programme vers le fichier. De même, la commande shell suivante redirige la sortie standard vers le fichier :

```
prog > fichier
```

En général, la sortie d'erreur reste attachée au terminal.

Ouverture/Fermeture du fichier La première tâche du programmeur qui souhaite ouvrir un fichier en lecture ou écriture est de créer cette représentation logique à partir du nom physique du fichier, cette opération est dépendante du système d'exploitation. En langage C, elle s'effectue à l'aide de l'opérarion :

```
FILE * fopen(char *nom, char *mode)
```

La variable `nom` est le nom physique du fichier et `mode` le mode d'ouverture qui peut être

`r`	`read` mode d'ouverture en lecture, le fichier ne pourra pas être modifié
`w`	`write` mode d'ouverture en écriture, le fichier sera écrasé
`a`	`add` mode d'ouverture en écriture, l'écriture se fera à la fin du fichier

Des erreurs peuvent se produire (droit d'accès inexistants, essai de lecture sur un fichier qui n'existe pas ...) En cas d'erreur, la fonction `fopen` renvoie la valeur `NULL`.

Lorsque l'on a terminé de travailler avec un fichier, il est conseillé de procéder à sa fermeture. Ceci se fait à l'aide de la commande :

```
int fclose(FILE *fp)
```

Cette fonction ferme le fichier associé à `fp`. En cas de fichier ouvert en écriture, cela provoque l'écriture physique du tampon d'écriture.

Opérations de lecture/écriture sur les fichiers

Les manipulations de fichier se font via la représentation logique :

int `getc(FILE *fp)`	le caractère suivant du flot de caractères associé au fichier de représentation `fp`
int `putc(int c, FILE *fp)`	écrit le caractère *c* sur le flot de caractères associé au fichier de représentation `fp`
int `fprintf(FILE *fp, char *format,...)`	écrit les arguments en suivant le format d'impression `format` sur le flot de caractères associé au fichier de représentation `fp`
int `fscanf(FILE *fp, char *format,...)`	lit une entrée en suivant le format d'impression `format` sur le flot de caractères associé au fichier de représentation `fp`

Ces fonctions se comportent de manière analogue à `getc`, `putc`, `printf` et `scanf`; elles travaillent de manière générale sur une représentation logique de fichier alors que `getc` et `scanf` travaillent spécifiquement avec le canal d'entrée standard `stdin` et `putc` et `printf` avec le canal de sortie standard `stdout`.

Diagnostic d'erreurs

La fonction `ferror` permet de tester si une erreur s'est produite dans la communication avec le fichier. La fonction `feof` permet de tester si la fin du fichier a été atteinte.

```
int ferror (FILE *fp)
int feof (FILE *fp)
```

4.6.6 Le cas du Fortran

En Fortran, l'interface de lecture ou d'écriture se fait via des *unités logiques* associées à des flots de données. Les unités logiques sont des numéros. Le clavier a pour unité 5 et l'écran a pour unité 6. Ces deux unités peuvent être notées `*` dans les commandes `read` et `write`.

Ouverture/Fermeture de fichier

L'ouverture d'un fichier se fait par la commande :

`OPEN(<const-entier>,FILE='<nom-de-fichier>',STATUS='<statut>')`

Le premier argument est l'unité logique associée au fichier. Le statut est optionnel, il indique si l'ouverture doit créer un nouveau fichier (état `NEW`) ou bien s'il doit exister (état `OLD`) ou bien s'il ne doit exister que le temps du programme (état `SCRATCH`), par défaut c'est le mode `UNKNOWN` qui est utilisé si le fichier existe, il est ouvert, sinon il est créé.... Des arguments supplémentaires permettent de gérer les cas d'erreur ou le mode d'écriture.

Le numéro d'unité est unique dans le programme. Tout fichier doit être ouvert avant de pouvoir effectuer des opérations de lecture ou d'écriture. La commande de base pour fermer le fichier est

`CLOSE(<const-entier>)`

On peut également revenir au début du fichier ouvert :

`REWIND(<const-entier>)`

Commande `READ`/`WRITE`

Les commandes de lecture et d'écriture `READ`/`WRITE` prennent comme argument une unité logique sur laquelle se fera la lecture et l'écriture, mais également un format, on peut indiquer également le point de programme où se brancher lorsque l'on atteint la fin de fichier. À la suite on indique une suite d'arguments à écrire ou de valeurs gauches à lire.

<instruction> `::=` WRITE `"("`*<unite>*`","`*<format>*`")"` *<expr>* { `","` *<expr>* }
<instruction> `::=`
READ `"("`*<unite>*`","`*<format>*`","END"="`*<label>*`")"` *<val-gauche>* {`","`*<val-gauche>*}

Le format peut être le symbole `*` auquel cas il sera calculé de manière automatique. On peut également indiquer un label (numéro) correspondant à une ligne portant ce label à laquelle apparaîtra un format. Ce format commance par le mot clé `FORMAT` puis contiendra une suite de chaînes de caractères et de commandes de conversion ayant le même rôle que dans le cas de la commande `printf`. Le format peut aussi être une chaîne qui sera calculée par le programme.

4.7 Bibliothèques

Le langage C propose des bibliothèques standardisées proposant de nombreuses fonctions utiles. Les principales sont :

`<string.h>`	Fonctions de manipulation de chaînes de caractères : concaténation, copie, comparaison, recherche de position de caractères ...
`<ctype.h>`	Fonctions de tests et de conversions sur les caractères, passage entre majuscule et minuscule, repérage des caractères alphabétiques, numériques ...
`<math.h>`	Fonctions mathématiques avancée : trigonométrie, exponentiel, racine ...
`<stdlib.h>`	Fonctions utilitaires : allocation de mémoire, conversions de caractères en nombres, génération de nombre aléatoire, tri, fonctions mathématiques élémentaires...
`<stdio.h>`	Fonctions sur les entrées-sorties : manipulation de fichiers, lecture, écriture
`<time.h>`	Fonctions de traitement de l'heure et des dates
`<limits.h>`	Les constantes définissant les valeurs max des types entiers
`<floats.h>`	Les constantes définissant les valeurs limites des types flottants

Bibliographie

[1] J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[2] B. W. Kernighan and R. Pike. *La Programmation en pratique*. Langages-Programmation. Vuibert Informatique, 2001.

[3] B. W. Kernighan and D. M. Ritchie. *Le langage C, Norme ANSI*. Langages. Dunod, 2ème edition, 2000.

[4] A.C. Marshall. *Fortran 90 Course notes*. The University of Liverpool, 1997. `http://www.liv.ac.uk/HPC/F90CourseIntro3DayHomePage.html`.

[5] M. Metcalf. *Effective Fortran 77*. Oxford Science Publications, 1985.

[6] M. Metcalf and J. Reid. *FORTRAN 90/95 explained*. Oxford Science Publications, 1996.

[7] J.B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.