Résumé

La parallélisation automatique de programmes Fortran est une technique de plus en plus répandue grâce aux nombreux travaux menés sur le sujet. En revanche, la parallélisation de programmes écrits en langages impératifs plus modernes n'est pas très connue. En particulier, le langage C pose des problèmes intéressants de détection de parallélisme que nous nous proposons d'explorer.

La première phase d'un paralléliseur automatique est l'analyse syntaxique du langage source. Ce rapport décrit la représentation intermédiaire générée par cette première phase.

Un programme traduit en représentation intermédiaire est un arbre sous forme d'une liste *Lisp* ce qui présente les avantages de la lisibilité et de l'adéquation aux passes ultérieures. Cette représentation est de très haut niveau, car très proche du texte source. Les types sont représentés sous forme de chemin d'accès à un type de base. Les instructions de contrôle et une grande partie des expressions sont traduites de manière immédiate. En revanche, les accès à la mémoire sont représentés de manière unifiée grâce à deux opérateurs que nous introduisons.

Mots-clefs : compilation, langage C, parallélisation automatique.

Abstract

Automatic parallelization of Fortran programs is becoming a common technique thanks to numerous works on these matters. On the other hand, parallelization of programs written in other, more modern, imperative languages is not well known. Particularly, the C language poses interesting problems of concurrency detection that will be the subject of future research.

The first step of an automatic parallelizer is the syntactic analysis of the source text. This report describes the intermediate representation generated by this step.

A program translated in this intermediate representation is a tree represented as a *Lisp* list. This has two advantages : legibility and a better fit to the next steps. This is a high level representation because it is close to the source text. Types are represented as access path to a basic type. Control statements and a large part of expressions are translated in a straightforward way. On the other hand, memory accesses are represented in a unified way with two new operators.

Keywords : compilation, C language, automatic parallelization.

Table des Matières

Chapitre 1

Introduction

La parallélisation automatique de programmes, c'est à dire la transformation et la compilation d'un programme écrit pour une machine séquentielle sur un ordinateur parallèle, est en passe de devenir une technique courante grâce aux efforts de recherche menés depuis de nombreuses années. Cependant, le langage Fortran reste la cible principale de ces efforts.

Le langage Fortran étant limité dans bien des domaines, il paraît intéressant d'étudier la parallélisation automatique d'un langage disposant de fonctionnalités supplémentaires. Nous nous proposons d'explorer les problèmes liés a la parallélisation du langage C. Certes moins répandu que Fortran, le langage C dispose néanmoins d'une bonne notoriété et rend accessibles au programmeur des classes de problèmes non numériques, tels que la compilation ou l'écriture de systèmes d'exploitation, en un mot le “logiciel de base”. Les problèmes soulevés par le langage C sont, entre autres, les notions de pointeurs et d'indirection, les structures de données complexes et les types, nettement plus évolués qu'en Fortran.

La première étape d'un paralléliseur est l'analyse syntaxique, c'est à dire la traduction d'un programme écrit en langage C en une forme directement exploitable par les phases ultérieures, à savoir l'analyseur sémantique et le générateur de code. Ce rapport décrit en détail cette forme intermédiaire et, implicitement, l'analyseur syntaxique chargé de délivrer cette représentation intermédiaire.

La représentation que nous proposons est un arbre parenthésé, c'est à dire une expression ou une suite d'expressions “à la Lisp”. Cette représentation a plusieurs avantages :

- elle est lisible,
- elle est particulièrement bien adaptée aux phases ultérieures de l'analyse qui seront vraisemblablement écrites en Lisp, et
- de par sa syntaxe très simple, elle peut être facilement lue par un programme rédigé dans un langage quelconque.

Cette représentation est de très haut niveau par rapport aux représentations traditionnellement utilisées par les compilateurs. Le problème n'est pas ici de générer immédiatement du code, mais d'analyser le texte source.

Ce rapport présente donc la représentation des déclarations, et par conséquent des types, élément fondamental de notre forme intermédiaire. Ensuite, nous présentons brièvement les structures de contrôle, puis nous terminons sur les expressions, en incluant les accès à la mémoire. Enfin, une annexe donne au lecteur l'occasion de se familiariser avec l'objet de ce rapport avec des exemples.

Chapitre 2

Représentation des déclarations

Les codes intermédiaires utilisés dans les compilateurs traditionnels étant généralement de bas niveau, c'est à dire proches d'une machine ou d'un modèle portable de machine, il n'y a plus de représentation des déclarations.

A l'inverse, notre représentation intermédiaire doit servir de base à l'analyse sémantique, donc doit comporter toutes les informations nécessaires à l'analyse du code, et en particulier les déclarations des structures de données qui peuvent donner des indications à l'analyseur.

Dans cette section, nous étudierons d'abord la déclaration des variables et des fonctions, puis la représentation des types, élément essentiel du langage que nous ne cesserons d'utiliser dans la suite de ce travail.

2.1 Déclaration des variables et des fonctions

Les déclarations de fonctions ou de variables sont très similaires, ce qui nous permet de les représenter de manière homogène. De plus, les variables locales sont définies de la même manière que les variables globales, ce qui réduit le nombre de cas particuliers à gérer.

2.1.1 Déclarations et définitions

Les enseignants du langage C aiment bien souligner la différence entre les déclarations et les définitions. Les premières déclarent le type des objets manipulés sans leur allouer de la place, alors que les secondes allouent de la mémoire. Par exemple :

```
extern int x ;                    /* déclaration */
int y ;                           /* définition */
```

Il est vrai que cette différence n'est pas toujours perçue clairement par le débutant, principalement parce que la syntaxe traite de la même manière les déclarations externes (déclarations) et les définitions.

Dans la suite, nous ne soulignerons plus cette différence. Nous parlerons toujours de déclarations, qu'elles définissent ou non des zones de mémoire. La syntaxe en représentation intermédiaire étant identique, à la présence de l'allocateur `extern` près, nous ne nous préoccuperons plus de distinguer déclarations et définitions.

2.1.2 En avance sur les types

Les types en langage C sont très complexes. La deuxième moitié de cette section 2 leur est consacrée. Cependant, nous en avons besoin pour élaborer des déclarations. Pour ne pas trop anticiper le sujet, nous nous contenterons d'utiliser le type entier, que nous représenterons par `(simple-type int)`. Pour les fonctions, nous utiliserons un schéma plus complexe que nous expliciterons le moment venu.

Les premiers exemples que nous donnerons comporteront souvent des doubles parenthèses. Cette construction est dictée par un souci d'homgénéité. Le contenu des parenthèses les plus externes est une liste, qui se trouve dans les cas les plus simples avoir un seul élément.

2.1.3 Principe des déclarations de variables

Une déclaration de variable est une association de quatre informations :

1. le nom de la variable,
2. sa classe d'allocation,
3. son type, et
4. sa valeur initiale s'il y a lieu.

Le nom de la variable est un identificateur simple.

Il peut paraître étrange de dissocier la classe d'allocation (par exemple `static`, `register`, `extern`) du type de la variable, alors que le langage C les mélange dans les déclarations. En fait, la classe d'allocation spécifie la zone dans laquelle la mémoire est allouée (on retrouve ici la différence entre déclaration et définition du paragraphe précédent), alors que le type indique comment est utilisée cette zone de mémoire. La classe d'allocation est donc un attribut de variable, alors que le type peut être un attribut de n'importe quel objet manipulé dans notre représentation intermédiaire. Il est donc nécessaire de dissocier ces deux informations.

La valeur initiale est la valeur spécifiée dans le texte source. Ce peut être une constante entière, une constante flottante, une adresse en mémoire, ou encore une valeur composée de plusieurs objets, lors des initialisations de tableaux ou de structures.

Une déclaration de variable regroupe ces quatre informations dans une liste dont le *car* est `vdecl`. La syntaxe est donc :

```
(vdecl identificateur
       type
       classe-d'allocation
       valeur-initiale-ou-nil )
```

Il est temps de prendre un exemple. La déclaration en C suivante :

```
register int x = 3 + 4 ;
```

est représentée dans notre forme intermédiaire par l'expression :

```
(vdecl x
       ((simple-type int))
       register
       7)
```

Si la valeur initiale est une liste de valeurs (initialisation d'un tableau ou d'une structure), elle est représentée par une liste au sens Lisp.

2.1.4 Principe des déclarations de fonctions

Une déclaration de fonction est une association de quatre informations :

1. le nom de la fonction
2. sa classe d'allocation,
3. son type (incluant ses paramètres et son type de retour), et
4. son corps s'il y a lieu.

La syntaxe est donc :

(`fdecl` *identificateur*
type
classe-d'allocation
corps-de-la-fonction-ou-nil)

Nous voyons immédiatement que la similitude est grande entre déclaration de variable et déclaration de fonction. Le corps de la fonction remplace la valeur initiale de la variable, et le type semble avoir changé.

Nous regroupons ces deux informations avec la syntaxe suivante :

((`fun` *type-de-retour*
spécification-des-paramètres))

Le type de retour est un type comme celui vu dans le paragraphe précédent. La spécification des paramètres est une liste de déclarations conformes à la syntaxe de la déclaration des variables, et dont le *car* est le symbole `proto`. Par exemple, la fonction :

```
static int close (int fd)
{
    ...
}
```

est représentée par la forme intermédiaire suivante :

```
(fdecl close
       ((fun ((simple-type int))
             (proto (vdecl fd
                           ((simple-type int))
                           nil
                           nil) ) ))
       static
       ... )
```

Nous reviendrons en détail sur ce type *fun*.

2.1.5 Les classes d'allocation

Nous avons déjà signalé la différence entre classe d'allocation et type. La première spécifie la zone dans laquelle l'objet est mémorisé, alors que le second indique la taille de l'objet (dans le cas d'une variable) et surtout la manière dont se font les accès à l'information. Cette distinction n'est pas évidente en langage C, car la syntaxe ne distingue pas les deux concepts dans les déclarations.

Précisons que si `typedef` est considéré traditionnellement comme une classe d'allocation, il ne s'agit là que d'un artifice syntaxique qui ne nous concerne pas. Nous verrons `typedef` dans le paragraphe 2.2.5.

Aux classes d'allocation traditionnelles, c'est à dire présentes en langage C, nous ajoutons la classe `global`. Précisons la signification de ces classes pour les variables et les fonctions.

Classes d'allocation pour les fonctions

Les classes d'allocation autorisées pour les fonctions sont :

- `static` : fonction locale au fichier compilé (nom non exporté)
- `global` : fonction globale au programme (nom exporté)

Signalons encore la classe `extern` indiquant simplement le type d'une fonction (et éventuellement de ses paramètres), même si celle-ci est définie dans le même fichier. Aucune zone n'est allouée.

Classes d'allocation pour les variables

Les classes d'allocation pour les fonctions sont extrêmement simples. En revanche, les variables sont divisées en deux groupes : les variables globales et les variables locales.

Les variables globales ont les classes d'allocation suivantes :

- `static` : variable locale au fichier (nom non exporté)
- `global` : variable globale au programme (nom exporté)

Ici encore, la classe `extern` indique que la variable est allouée ailleurs.

Les variables locales ont les classes d'allocation suivantes :

- `auto` : variable simple placée dans la pile
- `register` : variable placée si possible dans un registre (sinon dans la pile)

- `static` : variable placée dans la zone de données commune (pas dans la pile)

Les paramètres ont une classe d'allocation `auto`, même s'ils sont déclarés `register`. En effet, dans ce cas, `register` ne fait que demander au compilateur de copier le paramètre de la pile (où il arrive) dans un registre (où il réside pendant l'activation de la fonction).

Enfin, les variables déclarées `extern` à l'intérieur d'une fonction sont considérées comme globales. Ceci est justifié par le fait que déclarer deux fois le même nom avec des types différents dans deux fonctions séparées constitue une erreur. Cela rejoint d'ailleurs le comportement des premiers compilateurs C, même si cela est contraire à la norme.

2.2 Représentation des types

2.2.1 Principe

Comme tous les langages de programmation dits "modernes", C autorise la déclaration et l'utilisation d'objets de types complexes, construits à partir de types de base et de constructeurs. Les types de base en C sont :

- les types entiers : `char`, `short int`, `int`, `long int` et
- le type nul : `void`.

A ces types de base viennent s'ajouter les structures et les unions, elles aussi définies par l'utilisateur, ainsi que les fonctions.

Si, en toute rigueur, ces catégories ne sont que des constructeurs, nous avons placé les structures et les unions à cette place par pur souci de réduire la complexité de la représentation intermédiaire. Par contre, une fonction est un objet pouvant être référencé au moyen d'un pointeur, et doit donc être considérée comme un type de base.

En revanche, sont considérés comme constructeurs :

- les tableaux d'objets, et
- les pointeurs sur des objets (variables ou fonctions).

Toute déclaration en C consiste à spécifier un type de base et une méthode d'accès au type. Par exemple, la déclaration classique en C :

```
char *argv [] ;
```

revient à déclarer que `argv` est un tableau de pointeurs sur des caractères.

Notre représentation des types explicite cette caractéristique. Une spécification de type est un *chemin d'accès* à un type de base, le chemin d'accès étant la liste des constructeurs (dans l'ordre de leur utilisation lors d'une référence) utilisés pour accéder au type de base. C'est en quelque sorte, la manière "légale" de référencer l'objet final.

Par exemple, le type de `argv` déclaré ci-dessus sera représenté par la liste :

```
((adecl nil) (pdecl) (simple-type char))
```

signifiant *tableau de taille inconnue*, puis *pointeur*, et enfin *type de base.*

2.2.2 Eléments d'une spécification de type

Plus spécifiquement, la représentation d'un type est une liste formée par :

1. la suite de constructeurs, et
2. le type de base

Constructeurs

Les constructeurs "classiques" sont :

- `(adecl` *constante-entière*`)` ou `(adecl nil)` : tableau de taille connue ou inconnue
- `(pdecl)` : pointeur

Les constructeurs `adecl` ne peuvent contenir qu'une seule dimension. Leur utilisation est ainsi similaire à celle des constructeurs `pdecl`.

A ces constructeurs classiques viennent s'ajouter les qualificateurs, ajout récent de la norme ANSI : il s'agit des `const` et `volatile`. Nous les représentons, tout naturellement, par :

- `(const)`
- `(volatile)`

Ainsi, la déclaration bientôt classique en C :

```
char const * const argv [] ;
```

signifiant que `argv` est un tableau de pointeurs constants sur des caractères constants est représentée par :

```
((adecl nil) (const) (pdecl) (const) (simple-type char))
```

Les qualificateurs viennent donc tout naturellement s'insérer dans les *chemins d'accès* au type de base.

Types de base

Les types de base sont ceux que nous avons examiné plus haut. Ils sont représentés par une liste contenant les mots-clefs utilisés en C. Par exemple :

- `(simple-type char)`
- `(simple-type unsigned long int)`
- `(void)`
- `(struct y)`

Les types de base sont définis par une liste dont le premier élément est la nature du type de base. Les différentes natures et leur syntaxe sont :

- `(simple-type` *liste-de-types*`)`
- `(struct` *nom-de-structure*`)` ou `(union` *nom-d'union*`)`
- `(void)`
- `(fun` *type-de-retour prototype*`)`

Le type `void` est utilisé comme type de base. En effet, on peut avoir un pointeur sur un objet `void`. Par cette méthode, la norme fournit un *pointeur générique*. Une opération de cœrcion de `void *` en *type-quelconque* `*` (et réciproquement) ne perd aucune information.

Les énumérations, déjà utilisées avant la norme, ne sont pas des types de base. Ces objets, très pratiques, sont en fait des entiers déguisés. Les énumérations n'ont pas accès à tous les opérateurs disponibles pour les entiers, les valeurs qu'elles peuvent prendre ne sont pas limitées, par le jeu des cœrcions, et certaines de ces valeurs sont représentées par une constante symbolique. Du point de vue du compilateur, les types énumérés n'apportent rien par rapport aux entiers. Nous transformerons donc systématiquement les énumérations en entier pour simplifier.

2.2.3 Structures et Unions

Par extension, nous avons considéré les types énumérés, les structures et les unions définis par l'utilisateur comme des types de base pour notre représentation intermédiaire des types. Nous utiliserons dans la suite de cet exposé, sauf mention explicite, le terme *structure* pour nommer ces trois objets.

Par exemple, un objet déclaré en C par :

```
struct element_liste chainons [10] ;
```

sera représenté dans notre forme intermédiaire par :

```
(vdecl chainons
       ((adecl 10) (struct element_liste))
       global
       nil)
```

Une structure non nommée explicitement par l'utilisateur se voit nommée implicitement par l'analyseur syntaxique afin de ne pas générer de cas particulier.

Une structure n'est pas rigoureusement un type de base. Nous aurions pû ôter toute notion de nom et expanser chaque utilisation du nom par la liste des champs composant cette structure. Cette solution n'a pas été retenue pour deux raisons :

1. d'une part, nous voulions garder une représentation intermédiaire simple. L'utilisation des noms rend nécessaire la déclaration des structures, mais allège considérablement l'écriture.
2. d'autre part, faire disparaître les noms ne permet plus de distinguer entre deux structures différentes, mais contenant les mêmes champs.

En revanche, il faut déclarer explicitement l'objet *structure*, et nous serons amenés à définir un nouveau type de déclarations en 2.3.

2.2.4 Type de base fun

Nous avons mentionné plus haut le type de base `fun`. Le moment est venu de détailler.

Attention : il n'est pas question de revenir sur la déclaration de fonction. Nous définissons uniquement un type *fonction*, ce qui est différent. Le type de base `fun` nous permet de déclarer des *objets de type fonction*, qui serviront à :

1. définir des pointeurs sur des fonctions, et
2. déclarer des fonctions

Dans ces conditions, le type de base `fun` ne définit que le type de retour et le type des paramètres si des prototypes conformes à la norme ANSI sont présents. Cet élément est donc soit une partie d'une spécification de type, soit une partie d'une déclaration de fonction contenant l'allocateur et le code de la fonction.

La syntaxe du type de base `fun` est :

(`fun` *type-de-retour*
prototype)

Le type de retour est une spécification de type ordinaire. Le deuxième champ, quant à lui, spécifie le mode de déclaration des paramètres (avec ou sans prototype), ainsi que les paramètres eux-mêmes.

Notons que les spécifications de types telles que nous les avons vues jusqu'ici étaient linéaires. Le type de base `fun` introduit la notion d'arbre de type, dans la mesure ou un type de base `fun` contient lui-même plusieurs spécifications de types.

Le prototype est la zone de déclaration des paramètres. Sa syntaxe est :

- (`proto` *liste-de-déclarations*), ou
- (`noproto` *liste-de-déclarations*).

La dernière forme indiquant l'absence de prototype conforme à la norme ANSI. Dans ce cas, la fonction est définie suivant la méthode traditionnelle.

Nous allons détailler ces deux formes dans les paragraphes suivants.

Déclarations de paramètres avec prototypes

Une déclaration de paramètres avec un prototype se comporte comme une suite de déclarations de variables. Par exemple, le type de la déclaration de la fonction suivante :

```
main (int argc, char *argv [])
{
    ...
}
```

est représenté par :

```
(fdecl main
       ((fun ((simple-type int))              ; type de retour
             (proto (vdecl argc               ; nom du paramètre
                           ((simple-type int)) ; type
                           auto               ; classe d'allocation
                           nil)               ; valeur initiale
                    (vdecl argv
                           ((adecl nil) (pdecl) (simple-type char))
                           auto
                           nil) ) ))
       global                                 ; classe d'allocation de main
       ...)
```

Dans ce cas, la classe d'allocation et la valeur initiales sont nulles.

Cas particulier des déclarateurs abstraits La norme prévoit l'utilisation des prototypes spécifiant les types des paramètres sans en spécifier le nom. Par exemple,

```
extern char *malloc (int) ;
```

Ces prototypes sont gérés de la même manière que précédemment, à la seule exception que l'identificateur n'est pas spécifié dans chaque déclaration de paramètre.

Cette définition de l'utilisation de la fonction se traduit donc en représentation intermédiaire par :

```
(fdecl malloc
       ((fun ((pdecl) (simple-type int))
             (proto (vdecl nil                ; nom non spécifié
                           ((simple-type int))
                           auto
                           nil) ) ))
       extern
       nil)
```

Cas particulier des fonctions sans paramètre Les fonctions sans paramètre sont déclarées avec des prototypes ne comprenant que le type `void`. Par exemple :

```
int getpid (void)
{
```

```
        ...
    }
```

est représentée par :

```
(fdecl getpid
       ((fun ((simple-type int))                ; type de retour
             (proto void)))                     ; pas d'argument
       global                                   ; classe d'allocation
       ...)
```

Dans le cas des prototypes, il serait envisageable d'utiliser `(proto)` à la place de `(proto void)`. Cependant, cela ne serait plus homogène avec les déclarations de fonction sans prototype et sans argument. Nous verrons d'autre part que ces dernières nécessitent la présence de `void`.

Cas particulier des fonctions à nombre d'arguments variable Les fonctions à nombre d'arguments variable sont représentées par un paramètre spécial (`ellipsis`) au lieu de la définition de paramètre habituelle. Notons que ce paramètre spécial ne peut apparaître qu'en dernière position dans la liste de paramètres du prototype. Par exemple, la déclaration :

```
extern int printf (char *, ...) ;
```

est traduite en :

```
(fdecl printf
       ((fun ((simple-type int))
             (proto (vdecl nil
                           ((pdecl) (simple-type char))
                           auto
                           nil) )
                    ellipsis) ))                ; nombre d'arguments variable
       extern
       nil)
```

Déclaration des paramètres sans prototype

Pour assurer la compatibilité avec les programmes existants, la norme autorise la définition de fonctions sans utiliser les prototypes. Le compilateur ne peut alors plus vérifier le nombre et le type des paramètres.

La déclaration des paramètres est alors faite avec `noproto`. Par exemple :

```
main (argc, argv)
int argc ;
char *argv [] ;
{
    ...
}
```

est représenté par :

```
(fdecl main
       ((fun ((simple-type int))              ; type de retour
             (noproto (vdecl argc
                             ((simple-type int))
                             auto
                             nil)
                      (vdecl argv
                             ((adecl nil) (pdecl) (simple-type char))
                             auto
                             nil) ) ))
       global                                 ; classe d'allocation
       ...)
```

Deux cas particuliers restent cependant à examiner : il s'agit des fonctions dont les paramètres ne sont pas (encore) connus et des fonctions sans paramètres.

Fonctions dont les paramètres ne sont pas connus Dans les programmes C antérieurs à la norme, il est très fréquent de rencontrer des lignes telles que :

```
extern char *malloc () ;
```

ayant pour effet de déclarer le type de retour de la fonction. En revanche, la liste de paramètres est vide, ce qui ne signifie pas que la fonction n'a pas de paramètre, mais que nous ne savons encore rien d'eux. Il faut une forme spéciale pour gérer ce cas :

```
(fdecl malloc
       ((fun ((pdecl) (simple-type char))
             (noproto) ))                     ; paramètres inconnus
       extern                                 ; classe d'allocation
       nil)
```

Notons qu'une fonction dont les paramètres ne sont pas connus est forcément une fonction définie sans prototype et avec la classe d'allocation `extern` (voir paragraphe 2.1.5).

Fonctions sans paramètre On ne peut savoir avec certitude qu'une fonction n'admet aucun paramètre que lorsqu'elle est définie (avec son corps). Nous représentons ces fonctions de manière analogue aux fonctions avec prototype. Par exemple :

```
int getpid ()
{
    ...
}
```

est représentée par :

```
(fdecl getpid
       ((fun ((simple-type int))             ; type de retour
             (noproto void) ))               ; pas de paramètre
       global                                ; classe d'allocation
       ...)
```

Notons ici que `void` est nécessaire pour distinguer les fonctions sans paramètre des fonctions dont les paramètres ne sont pas connus, décrites au paragraphe précédent.

2.2.5 Typedef

Nous avons jusqu'ici omis de parler des types définis par l'utilisateur au moyen de `typedef`. Si ce mot-clef et ses implications sont extrêmement difficiles à analyser du point de vue syntaxique, il ne pose plus aucun problème dans la représentation intermédiaire. En effet, toute notion de type défini par l'utilisateur a disparu : le type a été expansé au moment de sa reconnaissance par l'analyseur syntaxique. Un exemple clarifiera les idées. Les lignes suivantes :

```
typedef int *pint ;           /* pint = pointeur sur entier */
pint *t1 [10] ;               /* tableau de 10 pointeurs sur pint */
pint (*t2)[10] ;              /* pointeur sur tableau de 10 pint */
```

sont représentées dans notre forme intermédiaire par :

```
(vdecl t1
       ((adecl 10) (pdecl) (pdecl) (simple-type int))
       global
       nil)
(vdecl t2
       ((pdecl) (adecl 10) (pdecl) (simple-type int))
       global
       nil)
```

2.3 Déclaration des modèles de structures et d'unions

Nous avons vu en 2.2.3 la déclaration des objets structurés. Nous avions mentionné la nécessité de déclarer le modèle. Un modèle est une suite d'associations nom-type. Ainsi, si nous avons la déclaration C suivante :

```
struct chainon
{
     int clef ;
     struct chainon *suivant ;
} ;

struct chainon *tete ;
```

La déclaration de `tete` dans notre représentation intermédiaire sera :

```
(vdecl tete
        ((pdecl) (struct chainon))
        global
        nil)
```

Cette déclaration suppose connu le modèle `chainon`. Le modèle est déclaré par :

```
(sdecl chainon
        (clef ((simple-type int)))
        (suivant ((pdecl) (struct chainon))) )
```

Tout objet structuré doit donc avoir un modèle, même si l'utilisateur ne le nomme pas explicitement. Par exemple, la déclaration suivante en langage C :

```
struct
{    char a ;
     int b ;
} var ;
```

alloue un objet nommé `var` et en même temps définit le modèle de structure. L'analyseur syntaxique doit alors procéder en deux temps :

1. déclaration du modèle avec un nom unique,

2. déclaration de `var` suivant le modèle juste déclaré.

Les `unions` sont déclarées avec `udecl` au lieu de `sdecl`, et ont une syntaxe et une représentation identique à celles des structures.

Chapitre 3

Structures de contrôle

Le langage C fournit des structures de contrôle assez riches. Dans ce chapitre, nous allons expliciter leur représentation intermédiaire.

3.1 Structure de bloc

La première structure de contrôle est le bloc, qui permet d'introduire plusieurs instructions là où le langage n'en attend qu'une seule. De plus, le langage C permet de déclarer de nouvelles variables à l'intérieur des blocs.

Nous avons choisi la classique expression Lisp `(prog ...)` pour représenter les blocs. Par exemple :

```
while (exp)
{
      i1 ;
      i2 ;
}
```

est représenté par :

```
(while exp
       (prog i1
             i2))
```

L'instruction Lisp `while` ne nécessite pas `prog`. Mais notre `while`, de même que toute notre représentation intermédiaire, n'est pas du code Lisp, mais un arbre représenté sous forme de liste. Il ne faut donc pas faire la confusion.

Le problème que nous n'avons pas résolu est celui de la représentation des déclarations. Nous éludons le problème en *aplatissant* toutes les déclarations au niveau de la fonction. Toutes les déclarations émises dans des blocs (à quelque niveau de profondeur) sont rejetées au début de la définition de la fonction, au même niveau que les variables locales. Ceci nous amène à considérer un mécanisme de renommage pour éviter l'homonymie.

La nécessité d'un tel mécanisme est mise en évidence par l'exemple suivant :

```
void exemple (void)
{
    int v ;                         /* bloc 1 */

    {
        char *v ;                   /* bloc 1.1 */

        {
            int v [5] ;             /* bloc 1.1.1 */
        }
    }

    {
        double v ;                  /* bloc 1.2 */
    }
}
```

Il y a dans cet exemple quatre déclarations de la variable `v` dans des blocs différents. Si nous voulons les déclarer toutes au même niveau, la première s'appellera `exemple-1-v`, la deuxième `exemple-1-1-v`, la troisième `exemple-1-1-1-v`, et la dernière `exemple-1-2-v`.

Les déclarations locales aux blocs sont une caractéristique très intéressante du langage C. Néanmoins, il s'agit là d'un artifice, certes simple, mais permettant une meilleure programmation. Notre représentation intermédiaire supprime cet artifice pour plus de simplicité.

3.2 Structures de boucle

Les structures de boucle disponibles en langage C sont les répétitions de type `while`, `do` ... `while` et `for`.

Leur syntaxe en forme intermédiaire est :

- (`while` *exp instruction*)
- (`dowhile` *exp instruction*)

- `(for (`*var init inc limite*`)` *instruction*`)`

Les structures `while` et `dowhile` sont traduites en représentation intermédiaire de manière immédiate.

La sémantique de la boucle `for` est celle de langages tels que Pascal ou Ada, et non pas celle de C. Partant de la constatation que les boucles `for` utilisées dans les programmes C sont le plus souvent des boucles `while` déguisées, et non pas les boucles `for` "classiques", on choisit de transformer les boucles `for` en boucles `while` selon le principe bien connu :

for (e_1 ; e_2 ; e_3)
    inst ;
	$\Longleftrightarrow$	e_1 ;
		while (e_2)
		{
		    inst ;
		    e_3 ;
		}

Dans ces conditions, l'analyseur syntaxique ne génère pas de boucle `for`, mais la syntaxe est spécifiée pour les passes ultérieures de l'analyse sémantique.

3.3 Structures de choix

Les structures de choix sont au nombre de deux : l'instruction `if` et l'instruction `switch`.

3.3.1 Le choix binaire

L'instruction `if` ne pose pas de problème particulier, et sa syntaxe en forme intermédiaire est :

`(if` *exp* i_1`)` ou `(if` *exp* i_1 i_2`)`

Comme en Lisp, `if` est soit binaire, soit ternaire. Remarquons que la structure de contrôle `if` est représentée de la même manière que l'opérateur ternaire `?:` (voir chapitre 4).

3.3.2 Le choix multiple

L'instruction `switch` est représentée par `selectq` suivant la syntaxe :

`(selectq` *exp*
    `(`*liste-de-valeurs* i_1`)`

$\vdots$
(*liste-de-valeurs* i_n))

Comme en Lisp également, le cas `default` de C est représenté par la liste de valeurs `t`.

La structure `selectq` de Lisp, et en particulier les *listes-de-valeurs* se prêtent bien à la représentation des cas enchaînés, tels que :

```
switch (c)
{
     case 1 :
     case 4 :
     case 7 :
          y = y - 1 ;
          break ;
     case 3 :
     case 6 :
     case 9 :
          y = y + 1 ;
          break ;
}
```

qui est représenté par :

```
(selectq ...
         ((1 4 7) ...)
         ((3 6 9) ...))
```

Le langage C permet les pires atrocités, illustrées par le segment de code suivant :

```
switch (c)
{
     if (i == 0)
     {
          case 1 :
               i = 1 ;
               if (j == 0)
                    default: break ;
     }
     case 2 :
          ...
}
```

Face à de telles aberrations qui, de toutes manières, poseront des problèmes évidents d'analyse sémantique, nous générerons la suite de tests et de branchements équivalents (voir paragraphe 3.4), laissant au restructureur de programmes le soin de remettre cela d'aplomb.

Dans ces conditions, nous traduirons par `selectq` toute suite de `case` terminée par une instruction `return`, `break` ou `continue` assurant une rupture de séquence, toute succession ininterrompue (autrement que par une ou plusieurs étiquettes) de `case` étant considérée comme un seul `case` à plusieurs valeurs.

Seules les instructions `return`, `break` ou `continue` peuvent garantir la rupture de séquence. Les appels de fonctions (tels que `exit` par exemple) ne le peuvent pas, le programmeur pouvant les redéfinir.

3.4 Etiquettes et Sauts

Les étiquettes sont représentées par :

(`label` *identificateur*)

et les sauts par :

(`goto` *identificateur*)

De plus, le langage C définit deux types de sauts non étiquetés : `break` et `continue`. Ils sont représentés par :

```
(break)
(continue)
```

Notons que `break` ne sert plus pour terminer les clauses de `switch` dans notre représentation intermédiaire : soit le `switch` est bien structuré, `selectq` terminant alors implicitement chaque clause, soit le `switch` est mal structuré et le programme est traduit par des `goto`, ne nécessitant plus de `break`.

3.5 Retour de fonction

L'instruction `return` permet la sortie prématurée d'une fonction avec éventuellement une valeur. Nous représenterons ces deux cas par :

- `(return)`
- `(return` *valeur*`)`

Chapitre 4

Représentation des expressions

La richesse des expressions est la caractéristique principale du langage C. Cette richesse constitue à la fois un désavantage et un avantage : désavantage, car elle nuit considérablement à la lisibilité, argument fréquemment utilisé contre C, et avantage car son pouvoir expressif est substantiellement plus grand que dans les langages traditionnels tels que Pascal.

Nous étudierons dans ce chapitre les deux constituants de base des expressions, c'est à dire les constantes et les identificateurs, puis nous verrons la manière de les combiner, c'est à dire les opérateurs.

4.1 Les constantes

Les constantes sont de trois types :

1. les constantes entières, c'est à dire les constantes décimales, hexadécimales et octales, ainsi que les constantes "caractère",
2. les constantes flottantes, de type `float`, `double` ou `long double`, et
3. les constantes pointeur, c'est à dire les "chaînes de caractères".

Nous ne nous préoccuperons pas ici du problème de la précision de la représentation des entiers et des flottants, car nous ne nous intéressons qu'à l'analyse sémantique du code. De toutes manières, l'importance croissante de l'arithmétique IEEE nous permet de simplifier considérablement tous ces problèmes.

4.1.1 Les constantes entières

Les constantes entières, quelle que soit leur base, sont représentées dans la forme intermédiaire par une constante Lisp décimale.

Par exemple, les quatre déclarations suivantes :

```
int a = 'A' ;
int a = 65 ;
int a = 0x41 ;
int a = 0101 ;
```

sont toutes représentées par :

```
(vdecl a
       ((simple-type int))
       auto
       65)
```

Les constantes non signées sont représentées par une constante positive, convertie explicitement en type *non signé* (voir paragraphe 4.3.2).

4.1.2 Les constantes flottantes

Les constantes flottantes sont représentées par des nombres flottants Lisp. La représentation des nombres flottants n'est pas le sujet de cette étude.

4.1.3 Les constantes pointeur

Ce qu'on appelle en langage C les "chaînes de caractères" sont en fait des adresses en mémoire où le compilateur range les caractères à la file, suivis du caractère nul (`\0`). La notation habituelle en C :

```
char *hexa = "0123456789abcdef" ;
```

implique deux actions pour le compilateur. Il doit d'abord placer les 17 caractères (y compris le caractère nul) en mémoire, puis générer le code d'affectation de cette adresse au pointeur déclaré.

Nous conservons dans la forme intermédiaire la dualité implicite de cette écriture. Par exemple, pour représenter la déclaration ci-dessus en supposant qu'il s'agit d'une variable locale, nous ferons :

```
(vdecl hexa
       ((pdecl) (simple-type char))
       auto
       "0123456789abcdef")
```

4.2 Les identificateurs

Examinons les deux lignes suivantes que l'on pourrait extraire de n'importe quel programme en langage C :

```
x = 3 ;
y = x ;
```

Ces deux utilisations du symbole `x` sont très différentes : la première place 3 à l'adresse identifiée par le symbole `x`, alors que la seconde lit ce qu'il y a à cette même adresse. Traditionnellement, on parle de *lhs* (*left hand side*) et de *rhs* (*right hand side*) pour différencier ces deux types d'utilisation.

Notre représentation intermédiaire va tenir compte de cette différence, tout en supprimant les cas particuliers qu'elle aurait pu entraîner : un identificateur dans cette représentation va s'évaluer comme son adresse.

Les identificateurs peuvent représenter des objets dans la pile (variables automatiques), dans la zone de données communes (variables globales) ou dans le code (fonctions).

4.2.1 Accès

Comment cela s'utilise-t-il ? Les utilisations d'identificateurs sont limitées à trois cas :

1. la référence,
2. l'affectation,
3. l'appel de fonctions

La référence, ou lecture d'une case mémoire, a la syntaxe suivante :

(`fetch` *type adresse*)

L'affectation, ou écriture dans une case mémoire, a la syntaxe suivante :

(`store` *type adresse expression*)

L'appel de fonctions a la syntaxe suivante :

(`fcall` *type adresse liste-d'arguments*)

Dans les trois cas, *adresse* est l'adresse où se fait la référence, c'est à dire dans la plupart des cas l'identificateur puisqu'il s'évalue en son adresse.

Le *type* est un type comme ceux que nous avons déjà vus. Il spécifie la manière dont se fait l'accès. Par exemple :

```
(fetch ((simple-type int)) x)
(fetch ((simple-type char)) x)
```

sont deux accès aux informations stockées à l'adresse x, différant par la taille de ce qui est lu. Pour l'appel de fonction, le *type* est le type du résultat.

L'opérateur `fetch` s'évalue en le contenu de la case mémoire adressée, l'opérateur `store` en l'expression affectée, et l'opérateur `fcall` en la valeur retournée par la fonction appelée.

Passons maintenant en revue les différents types d'objets auxquels les identificateurs nous donnent accès.

4.2.2 Les variables scalaires

Les variables scalaires sont les plus simples. Connaissant les syntaxes décrites dans le paragraphe précédent, les lignes de C suivantes :

```
x = 3 ;
y = x ;
```

s'écrivent, en supposant que les deux variables x et y sont du type `int` :

```
(store ((simple-type int)) x 3)
(store ((simple-type int)) y (fetch ((simple-type int)) x))
```

Le mot-clef `store` est un opérateur, c'est à dire que les exemples ci-dessus sont des expressions renvoyant la valeur affectée, comme naturellement l'affectation en C ou en Lisp (fonction `setq`).

Nous n'avons pas soulevé le problème des variables qualifiées `register`. Le langage C n'autorise pas l'extraction de l'adresse de ces variables. Le générateur de code devra donc considérer les `store` et `fetch` sur ces variables comme des cas particuliers. Mais rappelons que le qualificateur `register` n'est qu'une indication pour le compilateur et non une obligation. Nous pouvons très bien, si cela devient gênant, décider d'ignorer ce qualificateur sans pour autant changer la sémantique du programme.

4.2.3 Les pointeurs

L'accès aux pointeurs

Grâce au principe de l'évaluation des identificateurs, nous pouvons utiliser les pointeurs sans introduire de nouvelle forme spéciale. Ainsi, par exemple, les lignes de code C ci-dessous :

```
char *p ;
p = 5 ;
```

s'écrivent dans notre représentation intermédiaire :

```
(store ((simple-type char))
       (fetch ((pdecl) (simple-type char)) p)
       5)
```

En pratique, en supposant que les pointeurs ont tous la même taille, il suffit que le type lors du `fetch` commence par `pdecl`.

Pour affecter une adresse à un pointeur, comme ci-dessous :

```
char *p, x ;
p = &x ;
```

il suffit de faire en représentation intermédiaire :

```
(store ((pdecl) (simple-type char))
       p
       x)
```

Prenons maintenant un exemple plus conséquent pour illustrer l'utilisation des pointeurs sur les fonctions. Soit le segment de code suivant :

```
extern char int_to_char (int) ;
char (*pfn)(int) = int_to_char ;
char x ;
x = (*pfn) (5) ;
```

Il sera représenté, y compris les déclarations, par :

```
(fdecl int_to_char
       ((fun ((simple-type char))
             (proto (vdecl nil ((simple-type int)) auto nil) )))
       extern
       nil)
(vdecl pfn
       ((pdecl) (fun ((simple-type char))
                     (proto (vdecl nil ((simple-type int)) auto nil) )))
       auto
       int_to_char)
(vdecl x
       ((simple-type char))
       auto
       nil)
(store ((simple-type char))
       x
       (fcall ((simple-type char))
              (fetch pfn ((pdecl) (fun ((simple-type char))
                                       (proto (vdecl nil
                                                     ((simple-type int))
                                                     auto nil)))))
              5))
```

Cet exemple plus complexe, surtout à cause des déclarations il faut le noter, montre bien l'utilisation générale des pointeurs.

L'arithmétique sur les pointeurs

L'accès par les pointeurs ne se limite pas aux adresses mémorisées. Le langage C permet de faire de l'arithmétique sur les pointeurs. Par exemple, si nous écrivons :

```
int *x ; *(x + 10) = 6 ;
```

nous affectons la valeur 6 à l'entier situé à l'adresse $x+40$, si nous supposons qu'un entier occupe 32 bits, c'est à dire 4 unités.

Pour représenter cette arithmétique, nous utilisons les opérateurs généraux de l'arithmétique tels que nous les verrons en 4.3. Nous utiliserons également l'opérateur `sizeof`, rendu nécessaire par le fait que l'analyseur syntaxique ne connaît pas les tailles des différents types. Etant donné ces opérateurs, nous traduisons les lignes de C ci-dessus par :

```
(store ((simple-type int))
       (add (fetch ((pdecl) (simple-type int)) x)
            (mul (sizeof ((simple-type int))) 10) )
       6)
```

Avec nos conventions, la représentation de l'opérateur `&` est particulièrement simple. La ligne `p = &a ;` est représentée par :

```
(store ((simple-type int)) p a)
```

4.2.4 Les tableaux

Le langage C autorise une très grande souplesse dans l'accès aux tableaux soit par indiçage, soit par pointeur. Par exemple, les deux accès ci-dessous sont strictement équivalents :

```
char tab [10], x ;
x = tab [5] ;
x = *(tab + 5) ;
```

Il est alors possible de représenter ces deux lignes par une représentation intermédiaire unique, à savoir :

```
(store ((simple-type char))
       x
       (fetch ((simple-type char))
              (add tab
                   (mul (sizeof ((simple-type char)))
                        5)) ))
```

Notons une différence de taille qui n'apparaît pas évidente lorsqu'on écrit en C : il s'agit des *faux tableaux*. Ce que nous appelons les *faux tableaux* (ou *vrais pointeurs*) sont les pointeurs utilisés comme si ils étaient des tableaux. Par exemple :

```
char tab [10], *p, x ;
p = tab ;
x = p [5] ;
```

Dans ce cas, l'accès au tableau nécessite une indirection supplémentaire, et la représentation intermédiaire de ces expressions doit en tenir compte. L'affectation à `x` se traduit par :

```
(store ((simple-type char))
       x
       (fetch ((simple-type char))
              (add (fetch ((pdecl) (simple-type char)) p)
                   (mul (sizeof ((simple-type char)))
                        5))))
```

Rappelons que l'on trouve très fréquemment ces *faux tableaux* dans les fonctions, lorsque l'on passe un tableau (c'est à dire l'adresse de son premier élément) ou un pointeur, et que la fonction traite ce paramètre comme si c'était un tableau comme dans l'exemple ci-dessous :

```
int pipe (int pipefd [])
{
     pipefd [0] = ...
}
```

Il apparaît ici que `pipefd` est en fait un pointeur que l'on utilise comme un tableau, et l'accès à `pipefd[0]` doit être réalisé au moyen d'une indirection supplémentaire.

4.2.5 Les structures et les unions

L'accès aux structures et aux unions pose deux problèmes distincts :

1. l'accès aux structures et aux unions dans leur totalité, et
2. l'accès aux champs de structures et d'unions.

L'accès aux structures et aux unions dans leur totalité est autorisé par bien des compilateurs récents, et est officialisé par la norme ANSI.

L'accès aux champs des structures et l'accès aux champs des unions sont deux problèmes très différents. Il s'agit de déterminer les adresses relatives des champs par rapport au début de l'objet : dans le premier cas, les champs des structures sont à des adresses à déterminer, alors que dans le deuxième cas, les champs des unions sont toujours à des adresses fixes.

Accès aux structures et unions dans leur totalité

Dans les deux cas, cependant, il faut pouvoir affecter une structure ou une union à une autre. C'est le cas des affectations, des passages de structures ou d'unions par valeur et des valeurs de retour de type structure ou union. Nous représentons donc cet exemple :

```
struct modele s1, s2 ;
s1 = s2 ;
```

par :

```
(store ((struct modele))
       s1
       (fetch ((struct modele)) s2))
```

Le cas des unions est identique.

Accès aux champs d'unions

Les champs des unions commencent tous à la même adresse. La seule chose qui diffère entre les deux affectations :

```
union
{
     char c ;
     int i ;
} u ;
u.c = 65 ;
u.i = 65 ;
```

est la taille de l'objet accédé. Nous représentons ces deux affectations, avec ce que nous avons déjà vu, par :

```
(store ((simple-type char)) u 65)
(store ((simple-type int)) u 65)
```

Là, l'union ne nécessite aucun traitement particulier.

Accès aux champs de structure

Le problème est ici plus complexe puisque les champs commencent à des adresses toutes différentes. Nous avons donc introduit un opérateur spécial, `field`, servant à obtenir l'adresse relative du champ dans la structure. Notons que ce n'est rien de plus qu'un entier à ajouter à une adresse, mais nous ne savons pas encore quelle valeur numérique cela représente, et en fait cette valeur est sans importance au moment de l'analyse sémantique.

Ainsi, pour l'exemple ci-dessous :

```
struct modele
{
     char c ;
     int i ;
} ;
int x ;
struct modele s ;

x = s.i ;
```

nous traduirons ceci en forme intermédiaire par :

```
(store ((simple-type int))
       x
       (fetch ((simple-type int))
              (add s (field modele c)) ))
```

Grâce à cette idée simple (les identificateurs s'évaluent comme leur adresse), nous avons donc résolu tous les problèmes d'accès à la mémoire.

4.3 Les opérateurs

Il serait fastidieux de décrire tous les opérateurs, car ceux-ci sont très proches de leur équivalent C. Nous invitons donc le lecteur à se référer à la table des opérateurs ci-après.

4.3.1 Les opérateurs absents

Certains opérateurs du langage C ne sont pas repris tels quels dans notre représentation intermédiaire, soit parce qu'ils ne sont qu'une combinaison d'opérateurs plus simples, soit parce qu'ils induisent trop de problèmes.

Le calcul-affectation

Ce n'est pas un opérateur particulier, mais au contraire d'une famille d'opérateurs : la classe *op*=. Cette absence ne nuit pas, comme le montre l'exemple ci-dessous. La ligne :

```
x += 4 ;
```

pourrait être traduite par :

```
(store ((simple-type int))
       x
       (add (fetch ((simple-type int)) x)
            4))
```

Mais l'exemple ci-dessous montre les limites de cette traduction simpliste :

```
*fn () += 4 ;
```

Le contenu de la case adressée par le résultat de la fonction `fn` est augmenté de 4. Dans ce cas, il faut utiliser une variable intermédiaire pour éviter une double évaluation de la fonction `fn` et préserver la sémantique du programme original. En supposant que la fonction `fn` renvoie un pointeur sur un entier, l'exemple ci-dessus est donc traduit en la séquence de code suivante :

```
tmp = fn () ;
tmp = *tmp + 4 ;
```

pour être finalement traduit en forme intermédiaire en l'expression suivante :

```
(comma (store ((pdecl) (simple-type int))
              tmp
              (fcall ((pdecl) (simple-type int))
                     fn
```

```
                  nil))
          (store ((simple-type int))
                 (fetch ((pdecl) (simple-type int))
                        tmp)
                 (add (fetch ((simple-type int))
                             (fetch ((pdecl) (simple-type int))
                                    tmp) )
                      4) ) )
```

Cette séquence utilise l'opérateur `comma` décrit en 4.3.2.

Dans les cas simples, la variable temporaire n'est pas nécessaire. L'analyseur syntaxique, sans changer la sémantique du programme, pourrait reconnaître ces cas simples et générer la séquence de code minimale.

Il est bien entendu que dans le cas d'opérations sur des pointeurs, l'opérande (entière) est multipliée par la taille de l'objet pointé, comme nous l'avons déjà vu en 4.2.3.

Les pré-incrémentation et -décrémentation

Le cas des opérateurs de pré-incrémentation et pré-décrémentation est similaire au précédent.

Ainsi, la ligne C :

```
a = ++x ;
```

est traduite en :

```
(store ((simple-type int))
       a
       (comma (store ((pdecl) (simple-type int))
                     tmp
                     x)
              (store ((simple-type int))
                     (fetch ((pdecl) (simple-type int))
                            tmp)
                     (add (fetch ((simple-type int))
                                 (fetch ((pdecl) (simple-type int))
                                        tmp) )
                          1) ) ) )
```

Les post-incrémentation et -décrémentation

Le cas des opérateurs de post-incrémentation et post-décrémentation est beaucoup plus complexe : il faut mémoriser une valeur, effectuer une opération et renvoyer la valeur mémorisée. Cela fait une séquence trop complexe et trop implicite. Nous avons choisi d'"aplatir" cette séquence : l'analyseur syntaxique, lorsqu'il rencontrera une telle expression, la décomposera en autant d'expressions simples que nécessaire. Prenons quelques exemples pour comprendre les différents cas.

L'affectation simple :

```
a = x++ ;
```

est traduite en :

```
a = x ;
x = x + 1 ;
```

La boucle :

```
while (x++ != 0)
     ...
```

est transformée en :

```
while (x != 0)
{
     x = x + 1 ;
     ...
}
x = x + 1 ;
```

L'appel de fonctions :

```
fn (x++) ;
```

est remplacé par :

```
tmp = x ;
x = x + 1 ;
fn (x) ;
```

Dans des cas d'utilisation dans des expressions complexes, telles que :

```
a = x++ + x++ + fn (x++) ;
```

l'analyseur syntaxique générera du code comme s'il s'était agi de :

```
tmp1 = x ;
x = x + 1 ;
tmp2 = x ;
x = x + 1 ;
tmp3 = x ;
x = x + 1 ;
a = tmp1 + tmp2 + fn (tmp3) ;
```

Ce dernier cas paraît bien complexe. Ceci est dû principalement à la complexité de l'expression originale. Notons d'ailleurs que ce type d'expressions n'a pas de valeur prévisible : la norme spécifie clairement que : "entre deux points d'une séquence, un objet doit voir sa valeur modifiée au plus une fois par l'évaluation d'une expression. De plus, la valeur antérieure doit être accédée uniquement pour déterminer la valeur à mémoriser. Ce paragraphe rend indéfinies des expressions telles que `i = ++i + 1` tout en autorisant `i = i + 1`".

Notons enfin que ce dernier exemple est bien périlleux puisque ni la norme, ni la tradition ne garantissent l'ordre d'évaluation des opérandes.

4.3.2 Les opérateurs de la représentation intermédiaire

La représentation intermédiaire contient la plupart des opérateurs standard du langage C. Les tables ci-dessous les résument. Nous décomposons les opérateurs en opérateurs arithmétiques (ceux qui utilisent des valeurs, qu'elles soient entières ou flottantes) unaires, binaires ou ternaires, et les opérateurs spéciaux (ceux qui utilisent autre chose que des valeurs).

L'opérateur "virgule"

L'opérateur "virgule" est typique du langage C. Il sert à combiner plusieurs expressions en une seule, c'est le triomphe de l'effet de bord ! Son intérêt se manifeste dans deux situations bien particulières :

1. dans les boucles de type `for`, pour la mise à jour de deux indices par exemple,
2. lorsque le programmeur est paresseux et ne veut pas introduire d'accolades pour mettre deux lignes dans un corps de boucle. Ce programmeur paresseux peut par exemple être le préprocesseur C pour simuler une fonction par une macro, ou de manière générale un générateur de code automatique.

Cet opérateur n'introduit aucune fonctionnalité supplémentaire. Le programme peut être transformé pour le supprimer, comme par exemple dans l'exemple suivant :

```
a = b, c ;
```

trauit en :

```
b ;
a = c ;
```

Toutefois, la suppression de ces effets de bord est une tâche plus facile à réaliser dans les passes ultérieures. La représentation intermédiaire conserve donc cet opérateur sous le nom `comma`. C'est le seul opérateur à arité variable.

Il est clair que notre analyseur syntaxique intervient *après* le préprocesseur `cpp`, ce qui fait que les définitions et utilisations de macros ne sont pas visibles pour lui.

Les opérateurs arithmétiques

Opérateurs unaires

Langage C	Représentation intermédiaire
`-`	`neg`
`~`	`bitnot`
`!`	`lognot`
`*`	`fetch` (voir 4.2.3)
`&`	inutile (voir 4.2)

Opérateurs binaires

Langage C	Représentation intermédiaire
`+`	`add`
`*`	`mul`
`/`	`div`
`-`	`sub`
`%`	`mod`
`&`	`bitand`
`|`	`bitior`
`^`	`bitxor`
`&&`	`logand`
`||`	`logior`
`<<`	`shr`
`>>`	`shl`
`==`	`eq`
`!=`	`ne`
`>`	`gt`
`<`	`lt`
`<=`	`le`
`>=`	`ge`

Opérateurs ternaires

Langage C	Représentation intermédiaire
`?:`	`if`

Les opérateurs spéciaux

Les opérateurs spéciaux sont les opérateurs utilisant au moins un objet qui n'est pas une valeur. Par exemple, `sizeof` demande un type, `=` nécessite un type, une adresse et une valeur. Voici leur liste :

Langage C	Représentation intermédiaire
`sizeof`	`sizeof`
`->`	`sfield` (voir 4.2.5)
`.`	`sfield` (voir 4.2.5)
`[]`	inutile (voir 4.2.4)
`()`	`cast`
`=`	`store` (voir 4.2.1)

sizeof L'opérateur `sizeof` prend en argument un type, et renvoie la taille occupée par un objet de ce type. L'opérateur `sizeof` du langage C autorise une expression comme argument. Notre représentation intermédiaire ne l'autorise pas. L'analyseur syntaxique doit remplacer ces expressions par leur type.

accès aux champs de structure Les accès aux champs de structure (opérateurs `.` et `->`) ont déjà été vus en 4.2.5. La différence entre ces deux opérateurs ne tient qu'à une indirection.

accès aux éléments d'un tableau Les accès aux éléments d'un tableau ont été vus en 4.2.4. Ils sont remplacés par des accès par des pointeurs.

cœrcion L'opérateur de cœrcion (*cast*) change le type d'une expression. Toutes les expressions sont typées dans notre représentation. Le type est en effet déduit des opérandes (c'est un type hérité). Une expression étant un arbre, le type remonte donc des feuilles vers la racine. Chaque opérateur (nœud de l'arbre) change le type de l'expression, mais `cast`, `fetch`, `store` et `fcall` le font explicitement.

affectation Nous avons vu en détail l'affectation dans 4.2.

Chapitre 5

Conclusion

La représentation que nous avons proposé dans ce rapport est très complète et recouvre presque tous les aspects du langage C. Seuls quelques points n'ont pas été abordé et sont donc candidats à des extensions futures. Il s'agit notamment du calcul et de la représentation des flottants, des champs de bits dans les structures, et des entiers non signés. Hormis ces points, tout programme se conformant à la norme ANSI peut être traduit dans notre forme intermédiaire.

Notons que cette représentation peut être étendue à d'autres langages. Fortran, par la simplicité de ses structures, se prête bien à une telle traduction. Il également possible de traduire d'autres langages impératifs traditionnels, tels que Pascal, par exemple, moyennant des aménagements pour des points délicats comme les fonctions et procédures imbriquées, et les environnements des fonctions transmises en paramètre.

Pour résumer, la représentation intermédiaire décrite dans ce rapport est de haut niveau car elle décrit directement en Lisp l'arbre équivalent au programme exprimé en langage C. Les types sont représentés de manière claire et homogène sous forme d'un chemin d'accès à un objet de base, complètement linéarisé et simple à parcourir pour les étapes ultérieures de l'analyse. Les accès à la mémoire sont les seuls opérateurs spécifiant un type, dégageant ainsi les autres opérateurs de cette responsabilité. Les structures de contrôle sont représentées de manière classique, de même que les expressions.

L'analyseur syntaxique est réalisé en C avec les outils de génération d'analyseurs (`lex` et `yacc`), ce qui a considérablement simplifié l'écriture et la mise au point. Tous les points essentiels sont traités. Des points particuliers comme la reconnaissance des effets de bord (post-incrémentation par exemple) ne sont pas encore traités, l'effort étant dorénavant placé sur les phases d'analyse sémantique. La conception d'un paralléliseur en passes distinctes permet d'envisager l'extension future de l'analyseur pour couvrir tous les aspects du langage C sans changer les passes ultérieures.

Chapitre 6

Bibliographie

[AHO86] Alfred Aho, Ravi Sethi, Jeffrey Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley, 1986

[AMM86] Zahira Ammarguellat, *Analyse syntaxique de programmes Fortran en vue de leur parallélisation*, Rapport Technique MASI No 2, Juin 1986

[HAR87] Samuel Harbison, Guy Steele, *C, A Reference Manual, second edition*, Prentice Hall, 1987

[JOH78] S. C. Johnson, *Yacc - Yet Another Compiler Compiler*, Bell Laboratories, New Jersey, 1978.

[KER77] Brian Kernighan, Dennis Ritchie, *The C Programming Language, first edition*, Prentice Hall, 1978

[KER88] Brian Kernighan, Dennis Ritchie, *The C Programming Language, second edition*, Prentice Hall, 1988

[LES78] M. E. Lesk, E. Schmidt, *Lex - A Lexical Analyzer Generator*, Bell Laboratories, New Jersey, 1978.

[SCH85] Axel Schreiner, George Friedman, *Introduction to Compiler Construction with Unix*, Prentice Hall, 1985

[STE84] Guy Steele, *Common LISP : The Language*, Digital Press, 1984

[TAW87] Nadia Tawbi, Alain Dumay, Paul Feautrier, *PAF : Un paralléliseur automatique pour Fortran*, Rapport MASI No 185, 1987

Annexe A

Exemples

Cette annexe présente deux exemples tirés de *The C Programming Language, Second Edition* de Brian W. Kernighan et Dennis M. Ritchie. Il ne s'agit pas de présenter tous les aspects de la représentation intermédiaire, mais de compléter les exemples donnés dans les chapitres précédents et d'illustrer la représentation intermédiaire sur quelques petits exemples réalistes.

A.1 La fonction "strcpy"

```
/*
 * Exemple extrait de Kernighan & Ritchie 2eme edition :
 *
 * Section 5.5 : character pointers and functions, page 105 :
 * deux versions differentes de strcpy.
 */

/* strcpy: copy t to s; array subscript version */
void strcpy_a (char *s, char *t)
{
    int i;

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

/* strcpy: copy t to s; pointer version 1 */
void strcpy_b (char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

En représentation intermédiaire, cela donne :

```
(fdecl _strcpy_a
  ((fun ((void))
        (proto (vdecl _strcpy_a-s ((pdecl) (simple-type char)) auto nil)
               (vdecl _strcpy_a-t ((pdecl) (simple-type char)) auto nil))))
  global
  (prog (vdecl _strcpy_a-1 ((pdecl) (simple-type int)) auto nil)
        (vdecl _strcpy_a-2 ((simple-type int)) auto nil)
        (vdecl _strcpy_a-1-i ((simple-type int)) auto nil)
        (store ((simple-type int)) _strcpy_a-1-i 0)
        (while (ne (store ((simple-type char))
                          (add (fetch ((pdecl) (simple-type char))
                                      _strcpy_a-s)
                               (fetch ((simple-type int))
                                      _strcpy_a-1-i))
                          (fetch ((simple-type char))
                                 (add (fetch ((pdecl) (simple-type char))
                                             _strcpy_a-t)
                                      (fetch ((simple-type int))
                                             _strcpy_a-1-i))))
                   0)
               (prog (comma (store ((pdecl) (simple-type int))
                                   _strcpy_a-1
                                   _strcpy_a-1-i)
                            (store ((simple-type int))
                                   _strcpy_a-2
                                   (fetch ((simple-type int))
                                          (fetch ((pdecl) (simple-type int))
                                                 _strcpy_a-1)))
                            (store ((simple-type int))
                                   (fetch ((pdecl) (simple-type int))
                                          _strcpy_a-1)
                                   (add (fetch ((simple-type int))
                                               _strcpy_a-2)
                                        1))
                            (fetch ((simple-type int)) _strcpy_a-2))))))

(fdecl _strcpy_b
  ((fun ((void))
        (proto (vdecl _strcpy_b-s ((pdecl) (simple-type char)) auto nil)
               (vdecl _strcpy_b-t ((pdecl) (simple-type char)) auto nil))))
  global
  (prog (vdecl _strcpy_b-3 ((pdecl) (pdecl) (simple-type char)) auto nil)
        (vdecl _strcpy_b-4 ((pdecl) (simple-type char)) auto nil)
        (vdecl _strcpy_b-5 ((pdecl) (pdecl) (simple-type char)) auto nil)
        (vdecl _strcpy_b-6 ((pdecl) (simple-type char)) auto nil)
        (while (ne (store ((simple-type char))
                          (fetch ((pdecl) (simple-type char))
                                 _strcpy_b-s)
                          (fetch ((simple-type char))
                                 (fetch ((pdecl) (simple-type char))
                                        _strcpy_b-t)))
                   0)
               (prog (comma (store ((pdecl) (pdecl) (simple-type char))
                                   _strcpy_b-3
                                   _strcpy_b-s)
                            (store ((pdecl) (simple-type char))
                                   _strcpy_b-4
                                   (fetch ((pdecl) (simple-type char))
                                          (fetch ((pdecl) (pdecl) (simple-type char))
```

```
                            _strcpy_b-3)))
       (store ((pdecl) (simple-type char))
              (fetch ((pdecl) (pdecl) (simple-type char))
                     _strcpy_b-3)
              (add (fetch ((pdecl) (simple-type char))
                          _strcpy_b-4)
                   1))
       (fetch ((pdecl) (simple-type char))
              _strcpy_b-4))
(comma (store ((pdecl) (pdecl) (simple-type char))
              _strcpy_b-5
              _strcpy_b-t)
       (store ((pdecl) (simple-type char))
              _strcpy_b-6
              (fetch ((pdecl) (simple-type char))
                     (fetch ((pdecl) (pdecl) (simple-type char))
                            _strcpy_b-5)))
       (store ((pdecl) (simple-type char))
              (fetch ((pdecl) (pdecl) (simple-type char))
                     _strcpy_b-5)
              (add (fetch ((pdecl) (simple-type char))
                          _strcpy_b-6)
                   1))
       (fetch ((pdecl) (simple-type char))
              _strcpy_b-6))))))
```

A.2 Le programme “pattern”

```
/*
 * Exemple extrait de Kernighan & Ritchie 2eme edition :
 *
 * Section 4.1 : basics of functions, page 69
 */

/*
 * Pour simplifier la lecture de l'exemple, <stdio.h> n'est pas inclus.
 * "getchar" est donc une fonction de la librairie et non une macro du
 * preprocesseur. EOF doit etre defini.
 */

/* #include <stdio.h> */
#define EOF     -1
extern int getchar (void) ;

#define MAXLINE 1000            /* maximm input line length */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";        /* pattern to search for */

/* find all lines matching pattern */
main()
{
    char line[MAXLINE];
    int found = 0;
```

```
    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline:  get line into s, return length */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex:  return index of t in s, -1 if none */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++,k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

En représentation intermédiaire, cela donne :

```
(vdecl _pattern
  ((adecl 5) (simple-type char))
  global
  "ould")

(fdecl _printf
  ((fun ((simple-type int))
        (noproto)))
  extern
  nil)

(fdecl _getchar
  ((fun ((simple-type int))
        (proto void)))
  extern
  nil)
```

```
(fdecl _main
  ((fun ((simple-type int))
        (noproto void)))
  global
  (prog (vdecl _main-1 ((pdecl) (simple-type int)) auto nil)
        (vdecl _main-2 ((simple-type int)) auto nil)
        (vdecl _main-1-line ((adecl 1000) (simple-type char)) nil nil)
        (vdecl _main-1-found ((simple-type int)) auto 0)
        (while (gt (fcall ((simple-type int))
                          _getline
                          (_main-1-line 1000))
                   0)
               (prog (if (ge (fcall ((simple-type int))
                                    _strindex
                                    (_main-1-line _pattern))
                             0)
                         (prog (fcall ((simple-type int))
                                      _printf
                                      ("%s" _main-1-line))
                               (comma (store ((pdecl) (simple-type int))
                                             _main-1
                                             _main-1-found)
                                      (store ((simple-type int))
                                             _main-2
                                             (fetch ((simple-type int))
                                                    (fetch ((pdecl) (simple-type int))
                                                           _main-1)))
                                      (store ((simple-type int))
                                             (fetch ((pdecl) (simple-type int))
                                                    _main-1)
                                             (add (fetch ((simple-type int))
                                                         _main-2)
                                                  1))
                                      (fetch ((simple-type int))
                                             _main-2))))))
        (return (fetch ((simple-type int))
                       _main-1-found))))

(fdecl _getline
  ((fun ((simple-type int))
        (proto (vdecl _getline-s ((adecl nil) (simple-type char)) auto nil)
               (vdecl _getline-lim ((simple-type int)) auto nil))))
  global
  (prog (vdecl _getline-1 ((pdecl) (simple-type int)) auto nil)
        (vdecl _getline-2 ((pdecl) (simple-type int)) auto nil)
        (vdecl _getline-3 ((simple-type int)) auto nil)
        (vdecl _getline-4 ((pdecl) (simple-type int)) auto nil)
        (vdecl _getline-5 ((simple-type int)) auto nil)
        (vdecl _getline-1-c ((simple-type int)) auto nil)
        (vdecl _getline-1-i ((simple-type int)) auto nil)
        (store ((simple-type int)) _getline-1-i 0)
        (while (logand (logand (gt (comma (store ((pdecl) (simple-type int))
                                                 _getline-1
                                                 _getline-lim)
                                          (store ((simple-type int))
                                                 (fetch ((pdecl) (simple-type int))
                                                        _getline-1)
                                                 (sub (fetch ((simple-type int))
                                                             (fetch ((pdecl) (simple-type int))
```

```
                                                            _getline-1))
                                              1)))
                           0)
                       (ne (store ((simple-type int))
                                  _getline-1-c
                                  (fcall ((simple-type int))
                                         _getchar
                                         nil))
                           (neg 1)))
               (ne (fetch ((simple-type int))
                          _getline-1-c)
                   10))
       (prog (store ((simple-type char))
                    (add (fetch ((adecl nil) (simple-type char))
                                _getline-s)
                         (comma (store ((pdecl) (simple-type int))
                                       _getline-2
                                       _getline-1-i)
                                (store ((simple-type int))
                                       _getline-3
                                       (fetch ((simple-type int))
                                              (fetch ((pdecl) (simple-type int))
                                                     _getline-2)))
                                (store ((simple-type int))
                                       (fetch ((pdecl) (simple-type int))
                                              _getline-2)
                                       (add (fetch ((simple-type int))
                                                   _getline-3)
                                            1))
                                (fetch ((simple-type int))
                                       _getline-3)))
                    (fetch ((simple-type int)) _getline-1-c))))
(if (eq (fetch ((simple-type int)) _getline-1-c) 10)
    (prog (store ((simple-type char))
                 (add (fetch ((adecl nil) (simple-type char))
                             _getline-s)
                      (comma (store ((pdecl) (simple-type int))
                                    _getline-4
                                    _getline-1-i)
                             (store ((simple-type int))
                                    _getline-5
                                    (fetch ((simple-type int))
                                           (fetch ((pdecl) (simple-type int))
                                                  _getline-4)))
                             (store ((simple-type int))
                                    (fetch ((pdecl) (simple-type int))
                                           _getline-4)
                                    (add (fetch ((simple-type int))
                                                _getline-5)
                                         1))
                             (fetch ((simple-type int))
                                    _getline-5)))
                 (fetch ((simple-type int))
                        _getline-1-c))))
(store ((simple-type char))
       (add (fetch ((adecl nil) (simple-type char))
                   _getline-s)
            (fetch ((simple-type int))
                   _getline-1-i) )
```

```
              0)
       (return (fetch ((simple-type int))
                      _getline-1-i))))

(fdecl _strindex
  ((fun ((simple-type int))
        (proto (vdecl _strindex-s ((adecl nil) (simple-type char)) auto nil)
               (vdecl _strindex-t ((adecl nil) (simple-type char)) auto nil))))
  global
  (prog (vdecl _strindex-8 ((pdecl) (simple-type int)) auto nil)
        (vdecl _strindex-9 ((simple-type int)) auto nil)
        (vdecl _strindex-10 ((pdecl) (simple-type int)) auto nil)
        (vdecl _strindex-11 ((simple-type int)) auto nil)
        (vdecl _strindex-12 ((pdecl) (simple-type int)) auto nil)
        (vdecl _strindex-13 ((simple-type int)) auto nil)
        (vdecl _strindex-1-k ((simple-type int)) auto nil)
        (vdecl _strindex-1-j ((simple-type int)) auto nil)
        (vdecl _strindex-1-i ((simple-type int)) auto nil)
        (store ((simple-type int)) _strindex-1-i 0)
        (while (ne (fetch ((simple-type char))
                          (add (fetch ((pdecl) (simple-type char))
                                      _strindex-s)
                               (fetch ((simple-type int))
                                      _strindex-1-i)
                               ))
                   0)
               (prog (comma (store ((simple-type int))
                                   _strindex-1-j
                                   (fetch ((simple-type int))
                                          _strindex-1-i))
                            (store ((simple-type int))
                                   _strindex-1-k
                                   0))
                     (while (logand (ne (fetch ((simple-type char))
                                               (add (fetch ((pdecl) (simple-type char))
                                                           _strindex-t)
                                                    (fetch ((simple-type int))
                                                           _strindex-1-k)
                                                    ))
                                        0)
                                    (eq (fetch ((simple-type char))
                                               (add (fetch ((pdecl) (simple-type char))
                                                           _strindex-s)
                                                    (fetch ((simple-type int))
                                                           _strindex-1-j)
                                                    ))
                                        (fetch ((simple-type char))
                                               (add (fetch ((pdecl) (simple-type char))
                                                           _strindex-t)
                                                    (fetch ((simple-type int))
                                                           _strindex-1-k)
                                                    ))))
                            (prog (comma (comma (store ((pdecl) (simple-type int))
                                                       _strindex-10
                                                       _strindex-1-j)
                                                (store ((simple-type int))
                                                       _strindex-11
                                                       (fetch ((simple-type int))
                                                              (fetch ((pdecl) (simple-type int))
```

```
                                                          _strindex-10)))
                                              (store ((simple-type int))
                                                     (fetch ((pdecl) (simple-type int))
                                                            _strindex-10)
                                                     (add (fetch ((simple-type int))
                                                                 _strindex-11)
                                                          1))
                                              (fetch ((simple-type int))
                                                     _strindex-11))
                                       (comma (store ((pdecl) (simple-type int))
                                                     _strindex-12
                                                     _strindex-1-k)
                                              (store ((simple-type int))
                                                     _strindex-13
                                                     (fetch ((simple-type int))
                                                            (fetch ((pdecl) (simple-type int))
                                                                   _strindex-12)))
                                              (store ((simple-type int))
                                                     (fetch ((pdecl) (simple-type int))
                                                            _strindex-12)
                                                     (add (fetch ((simple-type int))
                                                                 _strindex-13)
                                                          1))
                                              (fetch ((simple-type int))
                                                     _strindex-13)))))
                   (if (logand (gt (fetch ((simple-type int))
                                          _strindex-1-k)
                                   0)
                               (eq (fetch ((simple-type char))
                                          (add (fetch ((pdecl) (simple-type char))
                                                      _strindex-t)
                                               (fetch ((simple-type int))
                                                      _strindex-1-k)
                                               ))
                                   0))
                       (prog (return (fetch ((simple-type int))
                                            _strindex-1-i))))
                   (comma (store ((pdecl) (simple-type int))
                                 _strindex-8
                                 _strindex-1-i)
                          (store ((simple-type int))
                                 _strindex-9
                                 (fetch ((simple-type int))
                                        (fetch ((pdecl) (simple-type int))
                                               _strindex-8)))
                          (store ((simple-type int))
                                 (fetch ((pdecl) (simple-type int))
                                        _strindex-8)
                                 (add (fetch ((simple-type int))
                                             _strindex-9)
                                      1))
                          (fetch ((simple-type int)) _strindex-9))))
        (return (neg 1))))
```

Annexe B

Syntaxe de la représentation intermédiaire

programme ::= declaration*

declaration ::= sdecl
| udecl
| vdecl
| fdecl

sdecl ::= (**sdecl** ident champ*)

udecl ::= (**udecl** ident champ*)

champ ::= (ident type)

type ::= (constructeur-ou-qualif* type-de-base)

constructeur-ou-qualif ::= qualificateur
| constructeur

qualificateur ::= (**const**)
| (**volatile**)

constructeur ::= (**pdecl**)
| (**adecl** **nil**)
| (**adecl** constante-entiere)

type-de-base	::=	(**struct** ident)
	|	(**union** ident)
	|	(**simple-type** simple-type)
	|	(**fun** type declaration-parametres)
	|	(**void**)
simple-type	::=	**char**
	|	**signed char**
	|	**unsigned char**
	|	**short**
	|	**signed short**
	|	**short int**
	|	**signed short int**
	|	**unsigned short**
	|	**unsigned short int**
	|	**int**
	|	**signed**
	|	**signed int**
	|	**unsigned**
	|	**unsigned int**
	|	**long**
	|	**signed long**
	|	**long int**
	|	**signed long int**
	|	**unsigned long**
	|	**unsigned long int**
	|	**float**
	|	**double**
	|	**long double**
declaration-parametres	::=	(**proto** liste-vdecl-proto)
	|	(**noproto** liste-vdecl-noproto)
	|	(**noproto**)
liste-vdecl-proto	::=	**void**
	|	vdecl-abstrait* **ellipsis**
	|	vdecl-concret* **ellipsis**
vdecl-abstrait	::=	(**vdecl nil** type **auto nil**)
vdecl-concret	::=	(**vdecl** ident type **auto nil**)
liste-vdecl-noproto	::=	**void**
	|	vdecl-concret*

vdecl	::=	(**vdecl** ident type allocateur-global valeur-initiale)
allocateur-global	::=	**static**
	|	**global**
	|	**extern**
valeur-initiale	::=	**nil**
	|	expression
fdecl	::=	(**fdecl** ident type allocateur-global corps)
corps	::=	(**prog** vdecl-local* instruction*)
vdecl-local	::=	(**vdecl** ident allocateur-local valeur-initiale)
allocateur-local	::=	**auto**
	|	**register**
	|	**static**
bloc	::=	(**prog** instruction*)
instruction	::=	while
	|	dowhile
	|	for
	|	if
	|	selectq
	|	goto
	|	label
	|	break
	|	continue
	|	return
	|	expression
while	::=	(**while** expression bloc)
dowhile	::=	(**dowhile** expression bloc)
for	::=	(**for** (ident expression expression expression) bloc)

if	::= (**if** expression bloc)
	| (**if** expression bloc bloc)
selectq	::= (**selectq** expression)
	| (**selectq** expression clause*)
clause	::= (**t** bloc)
	| ((constante$^+$) bloc)
goto	::= (**goto** ident)
label	::= (**label** ident)
break	::= (**break**)
continue	::= (**continue**)
return	::= (**return**)
	| (**return** expression)
expression	::= constante
	| ident
	| (op1 expression)
	| (op2 expression expression)
	| (op3 expression expression expression)
	| (**fcall** type expression (expression*))
	| (**store** type expression expression)
	| (**fetch** type expression)
	| (**cast** type expression)
	| (**sfield** ident ident)
	| (**sizeof** type)
	| (**comma** expression*)
op1	::= **neg**
	| **bitnot**
	| **lognot**
op2	::= **add**
	| **mul**

$\mid$ **div**
$\mid$ **sub**
$\mid$ **mod**
$\mid$ **bitand**
$\mid$ **bitior**
$\mid$ **bitxor**
$\mid$ **logand**
$\mid$ **logior**
$\mid$ **shr**
$\mid$ **shl**
$\mid$ **eq**
$\mid$ **ne**
$\mid$ **gt**
$\mid$ **lt**
$\mid$ **ge**
$\mid$ **le**

op3 ::= **if**