Projet de Compilation (Master 1 2005/2006)
Revision: 17

Roberto Di Cosmo

Date: 2005-10-06 17:00:17 +0200 (jeu, 06 oct 2005)

Table des matières

1 Organisation : un projet complet **2**
- 1.1 Analyse . . . 2
- 1.2 Coeur et backend . . . 2

2 Le langage Ctigre. **3**
- 2.1 Syntaxe : . . . 3

3 Remarques sur quelques restrictions aux programmes CTigre **4**
- 3.1 Définitions recursives . . . 4
- 3.2 Champs des enregistrements . . . 5

4 Quelques remarques sur la sémantique **5**

5 Librairie standard **5**

6 Phases du projet **6**
- 6.1 Phase 1 : analyse sémantique et calcul des attributs . . . 6
  - 6.1.1 Typage (cette partie est optionnelle) . . . 6
  - 6.1.2 Attributs pour la génération de code . . . 7
- 6.2 Phase 2 et 3 : code intermédiaire et génération de code pour la machine cible MIPS R2000 8
- 6.3 Tests . . . 8

7 Extensions possibles **8**

A Définition de Ctigre en BNF **9**
- A.1 Grammaire BNF du langage Ctigre . . . 10

B Exemples de programmes Ctigre légaux **11**
- B.1 Factoriel . . . 11
- B.2 Vecteurs . . . 11
- B.3 Types récursifs . . . 11
- B.4 Un exemple plus complexe : la fusion de deux listes . . . 12

Le projet.

Le but de ce projet est de réaliser un compilateur pour un langage de programmation fictif, mais suffisamment réaliste pour montrer toutes les difficultés que l'on peut rencontrer dans un compilateur moderne.

1 Organisation : un projet complet

On acceptera des groupes de *au plus* 3 personnes, et dans le rapport il devra être clairement indiqué qui a fait quoi. Mais on supposera que chaque membre d'un groupe est au fait de tout le projet, et pourra répondre à des questions sur tout le projet.

Les soutenances auront lieu dans le courant du mois de Décembre 2005 : il faudra remettre *au plus tard le mardi 13 Décembre 2004 à 12h00* par courrier électronique à `roberto@dicosmo.org`, `pierre.letouzey@pps.jussieu.fr` et `jch@pps.jussieu.fr` votre projet accompagné d'un bref rapport (pas plus de 10 pages), qui détaillera vos choix pour chacune des phases, et éventuellement les limitations de votre compilateur par rapport à ce qui était demandé ; **l'ensemble du code source du compilateur ne constitue pas un rapport en lui même**.

Le projet est divisé en trois parties assez indépendantes :

1. réalisation du front-end : analyse lexicale et syntaxique, production de l'arbre de syntaxe abstraite
   cette partie vous est fournie dans les fichiers de support
2. réalisation du coeur : de l'arbre de syntaxe abstraite à la production de code intermédiaire linéarisé, et notamment

   typage vérification des types **ceci n'est pas demandé pour le 13 Décembre**

   calcul des attributs level et offset

   traduction en code intermédiaire

   transformation du code intermédiaire jusqu'à la linéarisation
3. réalisation du back-end : du code intermédiaire à la génération de code assembleur, et notamment

   sélection d'instructions conversion de code intermédiaire en assembleur abstrait

   allocation des registres pour le 13 Décembre, on vous demande seulement l'allocation naïve

   chaîne de production compléter le tout en produisant un seul fichier assembleur qui inclut le `runtime.s` et qui peut être directement exécuté sous `spim` ou `xspim`

1.1 Analyse

La tâche
La première partie est composée de l'analyseur lexical et l'analyseur syntaxique aptes à reconnaître un programme Ctigre. Ces analyseurs *vous sont fournis* sous la forme du code ML produit par OcamlLex et OcamlYacc.

Ils utilisent le type `Absyn.exp` des arbres de syntaxe abstraite de Ctigre, retournent un arbre de syntaxe abstraite en cas de reconnaissance réussie, et impriment un message d'erreur sinon.

1.2 Coeur et backend

La tâche
Écrivez le coeur et le backend d'un compilateur pour programmes Ctigre, qui utilisera comme front-end celui qui vous a été fourni.
En particulier, votre compilateur doit :

- vérifier le typage (cette partie est optionnelle)
- décorer l'arbre syntaxique avec les informations sur `level` et `offset`
- produire le code intermédiaire arborescent
- linéariser le code
- sélectionner les instruction pour produire l'assembleur MIPS
- effectuer l'allocation des registres

Ce qu'il faut rendre
On vous demande de nous fournir une archive `.tar.gz` produite à l'aide des outils `tar` et `gzip`, contenant :

- un fichier `Makefile`
- un fichier `main.ml`
- les autres fichiers source du projet

L'invocation de `make` sans arguments doit produire un fichier `main`, binaire exécutable OCaml, qui lit sur l'entrée standard un fichier source CTigre, le compile jusqu'à un certain point, et imprime le résultat sur la sortie standard. Pour cela, il devra reconnaître sur la ligne de commande au moins les options :

`-ast` imprime l'AST en utilisant la fonction `PrintAbsyn.print` définie dans le fichier `prabsyn.ml` qui vous a été fourni
`-lo` imprime l'AST avec level et offset
`-int` imprime le code intermédiaire arborescent
`-lin` imprime le code intermédiaire linéarisé
`-rawasm` imprime l'assembleur brut
`-asm` imprime l'assembleur après allocation des registres

Le projet devra être envoyé par courrier électronique sous la forme d'un archive `tar.gz` à la fois à `roberto@dicosmo.org`, `pierre.letouzey@pps.jussieu.fr`, et `jch@pps.jussieu.fr`.

Conseils Vérifiez que ce que vous envoyez est bien compilable : une fois produit le `.tar.gz`, allez dans un répertoire vierge, décompactez l'archive, et faites `make clean` puis `make` pour vous assurer que tout se passe bien.
Pensez à concevoir un jeu de tests pour votre compilateur.

2 Le langage Ctigre.

Le lange choisi est *CTigre*, une variante du langage *Tigre* défini par Andrew Appel dans son livre "Modern compiler implementation in {Java,C,ML}"[1]. La syntaxe a été rendue plus proche de celle de OCaml (comme suggéré par Jean-Jacques Lévy).

Il s'agit d'un langage impératif qui a quelque traits semblables à C (comme le fait que tout expression du langage, même une affectation, a une valeur), et quelques traits semblables à Pascal (déclarations de fonctions locales, typage fort des expressions).

Dans ce langage, il est possible pour le programmeur de :
- définir des fonctions locales, avec portée statique des identificateurs comme dans Pascal ou Ocaml
- définir des fonction mutuellement récursive
- définir des types utilisateur, à partir des types de base "string", "char" et "int" à l'aide d'enregistrements et vecteurs
- définir des types locales (avec portée statique)
- définir des types mutuellement récursifs

On donnera dans la suite quelque exemple de programme Ctigre.

2.1 Syntaxe :

Un programme Ctigre est tout simplement une expression, mais dans ce langage une expression peut prendre maintes formes :

[1] Oui, trois version du même livre, une pour chacun des 3 langages entre accolades !

une expression de base : un identificateur, une constante, ou une expression plus complexe enclose entre parenthèse (ou bien entre les mots clefs 'begin' et 'end')

une expression qui dénote un objet de type complexe : il s'agit dans ce langage de vecteurs, comme dans le cas de

```
monTypeDeVect [300] of "z"
```

qui est un vecteur de 300 chaînes de caractères initialise a `"z"`, correspondant au type de vecteur `monTypeDeVect` qui doit être déclaré avant.
Ou alors d'enregistrements, comme dans le cas

```
monTypedEnregistrement {premierchamp=3;deuxiemechamp="a"}
```

qui est un enregistrement avec deux champs, un entier et une chaîne de caractères, correspondant au type d'enregistrement `monTypedEnregistrement` qui doit être déclaré avant.

une expression dite "ennaire" , ce qui peut être :
- un appel de fonction, comme `fact(3)` ou `pgcd(x,y)`
- une opération unaire ou binaire sur une expression (comme `2+3` ou `- fact(3)` ou `5>=2`)
- une "valeur gauche" (l-value en anglais), i.e. tout ce qui peut recevoir une affectation : dans Ctigre, c'est un identificateur ou une composante d'un objet de type complexe (comme `v[3]` ou `a.premierchamp`, ou `m[2][4]` etc.)
- une affectation d'une expression a une valeur gauche

enfin, on a les expressions dites de "sequençage" , vu qu'elles permettent de mettre ensemble plusieurs expressions. On retrouve :
- la concaténation d'expression, comme dans `v:=4; v:=v+3`
- la conditionnelle comme dans `if x > y then 3` ou `if x=y then 2 else 4`
- la boucle `for` et la boucle `while`

Mais on retrouve aussi deux constructions qui permettent de définir des types ou des expressions locales :
- la forme `type t1 = ... and tn = ... in exp` déclare les types `t1` ... `tn` dont la portée est limitée à l'expression `exp` (ces déclarations peuvent être mutuellement récursives)
- `let id1= ... and idn = ... in exp` déclare les identificateurs `t1` ... `tn` dont la portée est limitée à l'expression `exp` (ces déclarations peuvent être mutuellement récursives)

Il nous reste á spécifier ce qu'est un identificateur et quelles sont les constantes que l'on accepte dans ce langage :

Identificateur : une séquence de caractères alphanumériques et de _

Constantes : les entiers, les chaînes de caractères délimitées par les guillemets, et les caractères

Enfin, on peut introduire des commentaires, même imbriqués, avec les mêmes conventions qu'en C (i.e. `/*` ouvre et `*/` ferme un commentaire). Une description formelle complète de la syntaxe de CTigre est donnée en appendice.

3 Remarques sur quelques restrictions aux programmes CTigre

3.1 Définitions récursives

Le front-end qui vous est fourni vous garantit qu'il n'y aura jamais de définitions mutuellement récursives de fonctions *et* valeurs en même temps dans l'AST que vous aurez à compiler.

Donc, un programme comme

```
let a:= foo(0)
and foo(x:int):int = if x=0 then 1 else foo(x-a)
in foo(3)
```

qui correspond pourtant à la BNF, sera rejeté par l'analyseur, et vous n'aurez donc pas à vous poser la question sur comment le compiler.

Cependant, vous êtes fort libres de vous poser la question de savoir si compiler un tel programme est vraiment aussi compliqué que cela paraît.

3.2 Champs des enregistrements

Pour vous permettre de compiler un programme CTigre sans utiliser l'information de type, il est nécessaire de supposer, comme en OCaml, que les noms des champs des enregistrements soient uniques pour chaque type enregistrement[2].

Donc, le programme suivant ne sera pas légal, même si le front-end le laissera passer quand même :

```
type intlist = {hd: int, tl: intlist} in
type tree ={key: int, children: treelist}
and  treelist = {hd: tree, tl: treelist} in
```

Vous pouvez, bien entendu, produire un erreur pendant la traduction si dans le programme source les étiquettes ne sont pas distinctes.

Le programmeur devra écrire plutôt

```
type intlist = {hd: int, tl: intlist} in
type tree ={key: int, children: treelist}
and  treelist = {treehd: tree, treetl: treelist} in
```

Quelques exemples de programmes CTigre sont donnés en appendice.

4 Quelques remarques sur la sémantique

Dans cette section, on ajoutera au fur et à mesure toute clarification nécessaire sur la sémantique du langage CTigre.

ordre d'évaluation des paramètres des fonctions : les paramètres des fonctions sont supposés évalués de gauche à droite

conditionnelles : une expression conditionnelle avec les deux branches est une expression et retourne donc une valeur ; une expression conditionnelle avec une seule branche est une commande, et donc le typeur ne laisserait pas passer un programme CTigre où une telle commande se retrouverait à l'intérieur d'une expression, comme printint(if 3 > 4 then 2) ; cependant, personne ne vous empêche de faire une traduction qui stipule qu'une conditionnelle avec une seule branche a la valeur 0.

5 Librairie standard

Le langage CTigre que vous compilez dispose d'une petite librairie standard de fonctions qui sont disponibles au programmeur. Mise à part toutes les opérations arithmétiques et logiques sur les types de base, on vous demande de compiler correctement les appels aux fonctions de librairie suivantes, pour lesquelles une implémentation vous est fournie dans le fichier `runtime.s` de support :

sortie :

- **print(s)** imprime la chaîne de caractères s
- **printint(i)** imprime l'entier i

[2] Attention : dans CTigre, cette restriction n'est motivée que par la volonté de séparer le typeur du reste du projet, alors que dans OCaml les raisons de la restriction sont bien plus profondes.

- **printfloat(f)** imprime le flottant f

entrée :

- **getchar()** lit un caractère en entrée (opération bloquante)
- **getstring(n)** lit une chaïne de caractères de longueur maximale n en entrée (opération bloquante)
- **readint()** lit un entier
- **readfloat()** lit un flottant

conversions :

- **ord(c)** le code ASCII du caractère c
- **char(i)** le caractère de code ASCII i
- **mkstring(c)** une chaîne de caractères contenant le seul caractère c

chaînes :

- **size(s)** renvoie la longueur d'une chaîne
- **concat(s1,s2)** la concaténation des chaînes $s1$ et $s2$
- **substring(s,first,n)** retourne la sous-chaîne de `s` de longueur `n` qui commence à la position `first`

6 Phases du projet

Le projet se déroulera en trois phases assez indépendantes.

6.1 Phase 1 : analyse sémantique et calcul des attributs

6.1.1 Typage (cette partie est optionnelle)

Effectuez une vérification de type sur l'arbre de syntaxe abstraite pour vous assurer que le typage est respecté (toute utilisation d'un identificateur respecte sa déclaration de type). En cas d'erreur, envoyez des messages d'erreur informatifs, en utilisant aussi l'information de type `Location.t`.

> Les hypothèses que nous avons faites sur le langage nous permettent de regarder la phase de typage comme une étape tout à fait indépendante du projet : le typeur prendra en entrée l'AST, le vérifiera, il lèvera une exception dans le cas d'un erreur de type, et ne fera rien s'il est bien typé.
>
> Cela signifie que vous pouvez réaliser toutes les phases du compilateur sans avoir écrit le typeur (tout simplement, le code produit pour des programmes mal typés produira des erreurs à l'exécution, mais vous n'avez pas besoin de l'information de type pour produire le code).

Dans CTigre l'égalité des types complexes n'est pas structurelle, mais définitionnelle. Autrement dit, si on trouve deux déclarations identiques d'un type enregistrement ou tableau, elles produisent deux types *différents*.

Ex :

```
type a = {n:int}
in let v1 := a {n=3}
   in type b = {n:int}
      in let v2 := b {n=3}
         in v1 = v2
```

donne erreur de type (a et b sont des types incompatibles).

Dans CTigre on autorise les définitions récursives de types, à condition que toute récursion passe à travers d'un constructeur de type (array ou record).

Donc ceci est illégal :

```
type a = a in ...
```

Mais ceci est correcte (quoique pas très utile) :

```
type a = array of a in ...
```

Aussi, la syntaxe autorise la définition de variables sans spécifier leur type, comme dans

```
let a := 3 in ...
```

Un programme CTigre légal permet ces abréviations seulement si le type de l'identificateur déclaré peut être déduit du contexte (comme dans l'exemple), mais pas dans les autres cas, comme celui qui suit (qui n'est pas légal)

```
let a := nil in ...
```

En effet, `nil` étant une constante qui appartient à tout type enregistrement, on ne peut pas déterminer le type de `a` si on ne le spécifie pas comme suit

```
let a : untypenregistrement := nil in ...
```

Vous pouvez utiliser pour les types la structure suivante :

```
module Types =
struct
  type unique = unit ref
  type ty =
            RECORD of (Symbol.symbol * ty) list * unique
          | NIL
          | INT
          | STRING
          | ARRAY of ty * unique
      | NAME of Symbol.symbol * ty option ref
      | UNIT
end
```

6.1.2 Attributs pour la génération de code

Décorez ensuite les noeuds de l'arbre contenant des déclarations d'identificateurs avec deux attributs

(level) le niveau d'imbrication de la fonction définissant l'identificateur.

Ex : dans

```
let f(i:int):int =
  let x:=i-1 in
  let g(j:int):int =
    let v:int := x+2 in
    let w:int := v*j
    in w
  in g(i-3)
in f(4)
```

la fonction f est au niveau 2, la fonction g est au niveau 3, donc la variable x de f est au niveau 2, alors que v et w sont au niveau 3.

(offset) le numéro d'ordre de la définition à l'intérieur de la fonction. Dans l'exemple précédent, v est à l'offset 1 et w á l'offset 2. Vous pouvez aussi utiliser, plutôt que le numéro d'ordre de la définition, le décalage exacte par rapport à FP dans le bloc d'activation.

Ces attributs seront utiles au moment de la génération de code pour traiter les pointeurs entre environnements (que vous utilisiez les liens statiques simples ou des optimisations comme les displays).

6.2 Phase 2 et 3 : code intermédiaire et génération de code pour la machine cible MIPS R2000

On vous demande, en traversant l'arbre, de générer du code intermédiaire comme vu en cours, ensuite de produire du code pour une machine cible a partir de ce code intermédiaire.

Pour les fonctions, vous devez *impérativement* utiliser les conventions d'appel qui vous sont données en cours. En particulier, la structure du bloc d'activation d'une fonction f est la suivante :

adresse plus grande

	in-arg n
	...
	in-arg 1
$FP \rightarrow$	lien statique
	var. loc. 1
	...
	var. loc. k
	place pour les temporaires
	place pour FP
	place pour ra
	place pour les autres registres...
	place pour les arguments des fonctions
$SP \rightarrow$	qu'on appelle...

adresse plus petite

Et le pointeru de pile SP n'est pas modifié par le corps de la fonction f.

> Attention : les conventions d'appel *ont changé* par rapport aux années précédentes. Pas la peine, donc, d'essayer de réutiliser un projet fait par vos prédécesseurs, cela se verrait tout de suite et votre note en souffrirait lourdement.

On a choisi pour ce projet d'utiliser comme machine cible le MIPS R2000, un processeur RISC pour lequel on dispose d'un simulateur efficace qui tourne en environnement Unix (et bien sûr Linux) et Windows.

Vous trouverez les exécutables `spim` *(version console) et* `xspim` *(version graphique) installés pour les TX, et vous trouverez la distribution complète, qui contient dans le dossier Documentation le manuel complet et la description de l'assembleur MIPS dans* `http://www.cs.wisc.edu/~larus/spim` (il y a aussi une copie locale sur les TX dans `~dicosmo/CompilTools/spim.tar.gz`).

6.3 Tests

Vérifiez avec une batterie de tests que votre compilateur est correctement mis en oeuvre : prêtez une attention particulière à la portée des identificateurs, au typage et aux fonctions récursives (ex : la factorielle) et mutuellement récursives.

Un jeu de tests est disponible en ligne sur la même page que ce projet, mais il n'est pas exhaustif.

7 Extensions possibles

Pour qui voudra, il sera possible de se poser la question de la mise en place d'extensions sophistiquées du langage, comme :
- fonctions d'ordre supérieur (fonctions qui acceptent des fonctions en paramètre, et éventuellement, mais c'est beaucoup plus difficile, et déconseillé, fonctions qui peuvent retourner des fonctions comme résultat)

- allocation efficace des registres (jusque là, dans le projet nous supposons que toute variable utilisée par une fonction et tout temporaire utilisé dans les opérations dans le programme est mémorisé dans le bloc d'activation et non pas dans un registre)
- garbage collector (glaneur de cellules : chaque enregistrement ou tableau de CTigre est alloué mais jamais desalloué, ce qui n'est pas raisonnable sans un glaneur de cellules).

Appendice

A Définition de Ctigre en BNF

Une définition formelles de la syntaxe de Ctigre peut être donnée en utilisant une grammaire en forme BNF comme celle qui suit, où les symboles terminaux sont entre `'  '`, alors que les autres symboles sont considérés non terminaux. On ne spécifie pas ident, string-literal, integer-literal et char-literal. On rappelle qu'en notation BNF on se permet des abréviations fort pratiques, que nous résumons dans le tableau suivant (où ϵ est la notation habituelle pour le mot vide) :

La notation BNF	abrège les productions	
$S \rightarrow [S']$	$S \rightarrow \epsilon$	$S \rightarrow S'$
$S \rightarrow S'^*$ ou aussi $S \rightarrow \{S'\}$	$S \rightarrow \epsilon$	$S \rightarrow S'S$
$S \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$	$S \rightarrow \alpha_1$ $\quad\ldots$	$S \rightarrow \alpha_n$

Enfin, pour des raisons historiques, dans les définitions en BNF on écrit

$$S \ ::= \ \alpha \text{ à la place de } S \rightarrow \alpha$$

Important : faites bien attention à distinguer les caractères $|$, $\{$,$\}$, $[$, $]$ de la méta-notation des caractères terminaux ayant la même forme : comme expliqué dans le cours, pour rendre claire dans ce qui suit cette différence, on écrira `[ exp ]` pour la notation en BNF qui indique 0 ou une occurrence de exp, et `'[' exp ']'` pour le terminal `[` suivi de l'expression exp et du terminal `]`.

A.1 Grammaire BNF du langage Ctigre

Voici donc la grammaire de CTigre en BNF. Bien entendu, cette grammaire est ambiguë à souhait et sert juste a formaliser notre intuition de la syntaxe du langage Ctigre : il faut travailler (beaucoup) pour obtenir à partir de celle-ci une définition satisfaisante pour OcamlYacc (et ce travail représente en soi un vrai projet, qui fait partie du cours de Compilation 1 en Licence).

```
programme ::=  expression

expression  ::=
  primary-expression
| construction-expression
| nary-expression
| sequencing-expression

primary-expression  ::=
   ident
|  constant
|  '(' expression ')'
|  'begin' expression 'end'

construction-expression  ::=
|  type-id '[' expression ']' 'of' expression
|  type-id '{' label '=' expression {',' label '=' expression } '}'

nary-expression  ::=
   ident '(' [ expression {',' expression } ] ')'
|  '-' expression  |  expression bin-op expression
|  l-value       |  l-value ':=' expression

sequencing-expression  ::=
   expression ';' expression
|  'if' expression 'then' expression [  'else' expression ]
|  'while' expression 'do' expression 'done'
|  'for' ident '=' expression direction expression 'do' expression 'done'
|  'type'  type-binding  { 'and' type-binding }  'in' expression
|  'let'  let-binding  { 'and' let-binding }  'in' expression

direction ::= 'to' |  'downto'

l-value ::= ident  |  l-value '.' label  |  l-value '[' expression ']'

type-binding  ::=  type-id '=' type-expression

type-expression   ::=
   type-id
|  '{' ty-fields-nonempty '}'
|  'array' 'of' type-id

let-binding ::=
   ident [':' type-id ] ':=' expression
|  ident '(' ty-fields ')' [':' type-id ] '=' expression

ty-fields  ::=  [ type-fields-nonempty ]
```

```
ty-fields-nonempty  ::=  ident ':' type-id  {',' ident ':' type-id }

bin-op  ::=  '+' | '-'  | '*' | '/'  | '=' | '<>'
          | '<' | '<=' | '>' | '>=' | '|' | '&'

constant ::= integer-literal | string-literal
           | char-literal | 'nil'

type-id  ::=  ident

label  ::=  ident
```

B Exemples de programmes Ctigre légaux

Voici quelques exemples de programmes légaux

B.1 Factorielle

Il est possible de déclarer des fonctions récursives.

```
/* Voilà un exemple simple de programme Ctigre: la factorielle */

let nfactor(n: int): int =
if  n = 0
then 1
else n * nfactor(n-1)
in printint(nfactor(10))
```

B.2 Vecteurs

Il est possible de déclarer des types utilisateur à partir de constructeurs de types prédéfinies comme les enregistrements ou les vecteurs.

```
/* Déclaration d'un type vecteur et d'une variable de ce type vecteur */

type  arrtype = array of int in
let   arr1:arrtype := arrtype [10] of 0
in    arr1
```

B.3 Types récursifs

Il est possible de déclarer des types utilisateur à partir de constructeurs de types prédéfinis comme les enregistrements ou les vecteurs, même de façon récursive et mutuellement récursive.

```
/* Définitions de type récursives */

/* une liste */
type intlist = {hd: int, tl: intlist} in
/* un arbre */
type tree ={key: int, children: treelist}
and  treelist = {treehd: tree, treetl: treelist} in
let lis:intlist := intlist { hd=0, tl= nil }
in  lis
```

B.4 Un exemple plus complexe : la fusion de deux listes

```
/* Et voici un exemple de programme plus complexe: le merge de deux listes
   lues sur l'entree standard */

type any = {any : int} in
type list = {first: int, rest: list} in
let buffer := 'x' in
let readlist() : list =
  let readaint(any: any) : int =
    let i := 0 in
    let isdigit(s : char) : int =
        ord(buffer)>=ord('0') & ord(buffer)<=ord('9')
    and skipto() =
          while buffer=' ' | buffer='\n'
            do buffer := getchar() done
    in skipto();
       any.any := isdigit(buffer);
       while isdigit(buffer)
         do i := i*10+ord(buffer)-ord('0'); buffer := getchar() done;
       i
  in
    let any := any{any=0}
    in let i := readaint(any)
     in if any.any > 0
         then list{first=i,rest=readlist()}
         else nil
in
 let merge(a: list, b: list) : list =
   if a=nil then b
   else if b=nil then a
   else if a.first < b.first
      then list{first=a.first,rest=merge(a.rest,b)}
      else list{first=b.first,rest=merge(a,b.rest)}
in
 let printlist(l: list) =
   let printint(i: int) =
     let f(i:int) = if i>0
            then (f(i/10); print(mkstring(chr(i-i/10*10+ord('0')))))
     in if i<0 then (print("-"); f(-i))
              else if i>0 then f(i)
                          else print("0")
     in
       if l=nil then print("\n")
       else (printint(l.first); print(" "); printlist(l.rest))
in
  print("Entrez une liste triee d'entiers positifs, ");
  print("terminez par un caractere qui n'est ni chiffre ni espace\n");
  buffer:=getchar();
  let list1 := readlist() in
  let list2 := (
    print("Entrez une liste triee d'entiers positifs, ");
    print("terminez par un caractere qui n'est ni chiffre ni espace\n");
    buffer:=getchar(); readlist())
```

```
in

  /* BODY OF MAIN PROGRAM */
 let merged:=merge(list1,list2) in
 print("Voici la fusion des deux listes en entree:\n");
 printlist(merged)
```