

Projet de Compilation (Master 1 { 2009/2010)

22 octobre 2009

Table des matières

1	Organisation	2
1.1	Un projet complet	2
1.2	Ce qu'il faut rendre	2
2	Le langage CTigre.	3
2.1	Syntaxe	3
2.2	Quelques restrictions	4
2.2.1	Définitions récursives	4
2.2.2	Champs des enregistrements	4
2.3	Bibliothèque de base	5
2.4	Quelques remarques sur la sémantique	5
3	Phases du projet	6
3.1	Phase 1 : analyse sémantique et calcul des attributs	6
3.1.1	Analyse lexicale & syntaxique	6
3.1.2	Typage (<i>optionnel</i>)	6
3.1.3	Attributs pour la génération de code	7
3.2	Phase 2 : code intermédiaire	8
3.2.1	Traduction vers le code intermédiaire arborescent	8
3.2.2	Linéarisation du code intermédiaire (<i>partiellement fournie</i>)	8
3.3	Phase 3 : génération de code pour la machine cible MIPS R2000	9
3.3.1	Sélection d'instruction	9
3.3.2	Allocation de registre (<i>seule la version naïve est obligatoire</i>)	9
3.3.3	Chaîne de production	9
3.4	Tests	9
3.4.1	Procédure automatique	9
3.4.2	Jeux de test	9
4	Extensions possibles	10
A	Définition de CTigre en BNF	11
A.1	Grammaire BNF du langage CTigre	11
B	Exemples de programmes CTigre légaux	13
B.1	Factorielle	13
B.2	Vecteurs	13
B.3	Types récursifs	13
B.4	Un exemple plus complexe : la fusion de deux listes	13

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

1.1 Un projet complet

Le projet est à faire par groupe de 2 personnes *maximum*. Vous devrez préciser très clairement dans le rapport la répartition du travail. Nous supposons cependant que chaque membre d'un groupe est au fait de tout le projet, et pourra répondre à des questions sur tout le projet lors de la soutenance. Les notes de projet seront individuelles.

Le projet est divisé en trois parties assez indépendantes, qui seront détaillées à la section 3. En voici l'idée générale :

Front-end : analyse lexicale et syntaxique, production de l'arbre de syntaxe abstraite ;

Cœur : de l'arbre de syntaxe abstraite à la production de code intermédiaire linéarisé ;

Back-end : du code intermédiaire à la génération de code assembleur.

Les deux premières parties sont à rendre au plus tard le **mercredi 25 novembre 2009**. Vous devrez rendre la troisième partie accompagnée d'un bref rapport (pas plus de 10 pages) la semaine du 6 janvier. La date exacte vous sera communiquée ultérieurement. Le rapport 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**. Les soutenances auront lieu également au début du mois de janvier.

1.2 Ce qu'il faut rendre

On vous demande de nous fournir une archive `.tar.gz` produite à l'aide de la commande `tar -czf`, contenant l'intégralité des sources du projet, y compris le code fourni.

L'invocation de `make` sans arguments doit produire un fichier `ctigre`, binaire exécutable, 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 au moins reconnaître sur la ligne de commande les options décrites à la section 3.3.3.

Afin de pouvoir automatiser les tests, le respect de ces instructions est obligatoire, notamment le nom des options de la ligne de commande. Vous pouvez cependant ajouter d'autres options.

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. Vous pouvez aussi effectuer la procédure de test automatique décrite à la section 3.4.

Attention : Même si l'énoncé ressemble beaucoup, les conventions d'appel, le code fourni ainsi que d'autres détails *ont changé* par rapport aux années précédentes. Vous devez impérativement suivre les conventions données et le code fourni. Il est formellement interdit de réutiliser du code des années passées.

2 Le langage CTigre.

Le langage choisi est *CTigre*, une variante du langage *Tigre* défini par Andrew Appel dans son livre *Modern compiler implementation in fJava,C,MLg*¹. Il s'agit d'un langage impératif qui a quelques traits semblables à C (comme le fait que toute 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 ;
- définir des fonctions *mutuellement récursives* ;
- définir des *types utilisateur* :
 - à partir des types de base "string", "char" et "int" ;
 - où à l'aide d'*enregistrements* et *vecteurs* ;
- définir des *types locaux* avec portée statique ;
- définir des *types mutuellement récursifs*.

Quelques exemples complets de programmes CTigre sont donnés en appendice.

2.1 Syntaxe

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

une expression de base : un identificateur, une constante, ou une expression plus complexe entre parenthèse ou 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 initialisées à "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 « n-aire », 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 à une valeur gauche.

¹Trois versions du même livre, une pour chacun des 3 langages entre accolades

enfin, on a les expressions dites de "séquençage", 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 trois constructions pour définir des expressions ou des types locaux :

- 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)
- `var 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)
- `function f1(...) = ... and fn(...) = ... in exp` déclare les fonctions `f1 ... fn` dont la portée est limitée à l'expression `exp` (ces déclarations peuvent être mutuellement récursives)

Il nous reste à 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 des guillemets, les caractères et la constante `nil` pour dénoter l'absence d'une structure complexe (fin de liste, arbre vide, ...).

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.

2.2 Quelques restrictions

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

2.2.2 Champs des enregistrements

Pour vous permettre de compiler un programme CTigre sans utiliser l'information de type, il est nécessaire de supposer que les noms des champs des enregistrements sont uniques. 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
```

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
```

2.3 Bibliothèque de base

Le langage CTigre que vous compilez dispose d'une petite bibliothèque 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 suivantes, pour lesquelles une implémentation vous est fournie dans le fichier de support `runtime.s` :

sortie :

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

entrée (opérations bloquantes) :

`getchar()` lit un caractère en entrée
`getstring(n)` lit une chaîne de caractères de longueur maximale *n* en entrée
`readint()` lit un entier

conversions :

`ord(c)` le code ASCII du caractère *c*
`chr(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*

Le fichier `runtime.s` contient également des fonctions qui ne peuvent pas être appelées depuis un programme CTigre, mais qui sont utiles pour le compilateur :

`malloc(n)` alloue *n* octets sur le tas et retourne l'adresse correspondante
`exit()` termine le programme proprement

2.4 Quelques remarques sur la sémantique

Cette section pourra être étendue au fur-et-à-mesure de vos questions 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.

3 Phases du projet

Le projet se déroulera en trois phases assez indépendantes.

3.1 Phase 1 : analyse sémantique et calcul des attributs

3.1.1 Analyse lexicale & syntaxique

Un analyseur lexical et un analyseur syntaxique vous sont fournis sous la forme du code OCaml produit par OCamlLex et OCamlYacc. Si la reconnaissance réussit, ils produisent un arbre de syntaxe abstraite dont le type est `Ast.rawexp`; sinon ils impriment un message d'erreur. Nous vous fournissons aussi une fonction d'impression de l'arbre de syntaxe abstraite dans le module `AstPrint`.

3.1.2 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.

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 l'on trouve deux déclarations identiques d'un type enregistrement ou tableau, elles produisent deux types *différents*. Par exemple, le programme :

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

doit donner une 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 :

```
var 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 ci-dessus, mais ne le permet pas dans les autres cas, comme l'exemple illicite suivant :

```
var 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

```
var 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
```

3.1.3 Attributs pour la génération de code

La première étape obligatoire dans ce projet est de décorer l'arbre de syntaxe abstraite avec notamment les informations de `level` et d'`offset`. L'arbre produit doit être du type `Ast.attrexp`. L'un des rôles de cette transformation est de remplacer les variables qui restent locales à une fonction par des temporaires – et de remplacer par des couples (`level`, `offset`) les variables qui s'échappent et doivent être allouées en pile.

(**level**) le niveau d'imbrication de la fonction définissant l'identificateur. Dans l'exemple suivant :

```
function f(i : int) : int =
  var x:=i-1 in
    function g(j:int):int =
      var v:int := x+2 in
      var 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` à l'offset 2.

Pour cette étape, il vous est notamment demandé de réaliser une analyse d'échappement, c'est-à-dire de n'allouer sur la pile que les variables qui le nécessitent.

La seconde décoration que vous devez accrocher à l'arbre de syntaxe abstraite est la distinction entre les appels aux fonctions prédéfinies par le runtime, et les appels aux fonctions définies dans un programme CTigre.

3.2 Phase 2 : code intermédiaire

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, ici MIPS R2000, à 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

\$fp !	in-arg n ... in-arg 1 lien statique
	var. loc. 1 ... var. loc. k place pour les temporaires place pour les registres à sauvegarder... ... dont \$fp \$sp ! ... dont \$ra

adresse plus petite

La première moitié du bloc (jusqu'au lien statique) doit-être allouée par la fonction appelante ; la seconde moitié doit-être alloué par la fonction appelé ; au moment de l'appel le registre **\$sp** pointe alors sur le lien statique.

Attention : comme décrit dans l'entête du fichier `runtime.s`, les fonctions de la bibliothèque standard n'ont pas besoin de lien statique. Dans le cas d'un appel à une fonction prédéfinie, la fonction appelante ne doit donc empiler que les arguments ; dans ce cas au moment de l'appel, le registre **\$sp** pointe sur le premier argument.

3.2.1 Traduction vers le code intermédiaire arborescent

Il vous est demandé de traduire l'arbre de syntaxe abstraite décoré et produit par le front-end vers le code intermédiaire arborescent. Ce code est décrit dans le module `ir.mli` (*pour Intermediate Representation*). Afin de tester cette étape de traduction, nous vous fournissons un interprète de code intermédiaire : le module `IrInterp` exporte une fonction `interp` de type `Ir.ir -> unit` et réalisant cette interprétation.

Cette étape doit notamment : traduire les expressions de séquencages de CTigre vers un langage avec branchement et saut – exhiber l'utilisation du lien statique et du registre **\$fp** – isoler les chaînes de caractères statiques qui seront déclarés dans la zone `.data`.

3.2.2 Linéarisation du code intermédiaire

Cette partie vous est en grande majorité fournie dans le module `Canon`. Le code intermédiaire linéarisé produit par ce module est décrit dans le fichier `lin.mli`.

Cette opération permet de transformer le code intermédiaire en forme arborescente vers une forme linéarisée plus propice à sélection d'instruction. Les principales différences entre ces deux codes intermédiaires sont décrites dans le fichier `lin.mli`.

3.3 Phase 3 : génération de code pour la machine cible MIPS R2000

3.3.1 Sélection d'instruction

Il vous est demandé de réaliser l'étape de sélection d'instruction, préalable à la génération de code assembleur MIPS. Pour cela, une représentation d'un assembleur MIPS abstrait contenant une infinité de registre vous est fournie par le type `MipsAsm.abstr_instr`.

3.3.2 Allocation de registre

Il vous est aussi demandé de réaliser un allocateur de registre. Seul un allocateur naïf, c'est-à-dire allouant tous les temporaires en pile, est obligatoire. Pour cette version, une représentation d'un assembleur MIPS simplifié vous est fournie par le type `MipsAsm.simpl_instr`.

3.3.3 Chaîne de production

Une fois tout ce travail accompli, et à partir du fichier `runtime.s` et de des fonctions d'impression d'assembleur présentées dans le module `MipsAsmPrint`, votre programme doit être capable de produire un code assembleur complet, et exécutable par `mars`.

Votre programme doit comprendre les options en ligne de commandes suivantes :

```
-ast      imprime l'AST ;
-attr     imprime l'AST avec attribut ;
-int      imprime le code intermédiaire arborescent ;
-intir    simule l'exécution du code intermédiaire arborescent avec à l'interpréteur fourni ;
-lin      imprime le code intermédiaire linéarisé ;
-rawasm   imprime l'assembleur abstrait (avec infinité de registre) ;
-asm      imprime l'assembleur après allocation des registres.
```

3.4 Tests

3.4.1 Procédure automatique

Un ensemble de programmes ctigre vous est fourni comme jeu de tests. Lors de la soutenance, votre compilateur sera testé automatiquement sur un jeu de tests beaucoup plus grand et qui ne vous sera pas fourni à l'avance. Vérifiez que votre compilateur est compatible avec la procédure automatique de test en lançant l'une des commandes suivantes :

```
sh test-exec.sh          nécessite l'émulateur spim
sh test-exec.sh -mars    nécessite l'émulateur mars
sh test-exec.sh -intir   utilise l'interprète de code intermédiaire
```

Ce script compile et exécute l'ensemble des fichiers suffixés par `.tg` qui sont contenus dans le répertoire `TESTS`. L'affichage produit par chaque programme est conservé dans un fichier nommé comme le fichier source et suffixé par `.log`. Cette sortie est comparée à la sortie attendue qui se trouve dans le fichier suffixé par `.out`. Si lors de la procédure automatique, un de ces programmes doit lire des données, celle-ci seront lues dans le fichier suffixé par `.in`.

Ce script affiche le nombre de tests réussis, et met en évidence ceux ayant échoués.

3.4.2 Jeux de test

Le jeu de test fourni étant minimaliste, vous devez établir un jeu de tests le plus complet possible pour vérifier que votre compilateur est correctement mis en œuvre : prêtez une attention particulière à la portée des identificateurs, aux fonctions récursives (ex : la factorielle) et mutuellement récursives, et aux fonctions internes accédant aux variables définies dans un niveau supérieur. Vous devez tester votre compilateur le plus souvent possible. Vous devez tester votre compilateur avec les jeux de tests écrits par les autres binômes. Votre jeu de test... sera vérifié avec notre compilateur.

4 Extensions possibles

Pour qui voudra, il sera possible de se poser la question de la mise en place d'extensions sophistiquées du langage, comme :

- convention d'appel de fonctions utilisant les registres `$a0` à `$a3` pour les quatres premiers arguments (faites attention en particulier aux arguments qui s'échappent)
- 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)
- back-end pour un autre processeur

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
$S \rightarrow [S']$	$S \rightarrow \epsilon \quad S \rightarrow S'$
$S \rightarrow S'^*$ ou aussi $S \rightarrow fS'g$	$S \rightarrow \epsilon \quad S \rightarrow S'S$
$S \rightarrow \alpha_1 j \dots j \alpha_n$	$S \rightarrow \alpha_1 \quad \dots \quad S \rightarrow \alpha_n$

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 $j, f, g, [,]$ 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 à 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 ; il fait partie du cours d'analyse syntaxique en 3ème année de 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 'f' label '=' expression f', ' label '=' expression g 'g'

nary-expression ::=
    ident '(' [ expression f', ' expression g ] ')'
| '-' 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 f 'and' type-binding g 'in' expression
| 'var' var-binding f 'and' var-binding g 'in' expression
| 'function' fun-binding f 'and' fun-binding g 'in' expression

direction ::= 'to' | 'downto'

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

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

type-expression ::=
    type-id
| 'f' ty-fields-nonempty 'g'
| 'array' 'of' type-id

var-binding ::=
    ident [ ':' type-id ] ':=' expression

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

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

ty-fields-nonempty ::= ident ':' type-id f', ' ident ':' type-id g

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.

```
/* Voila un exemple simple de programme CTigre: la factorielle */
function 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.

```
/* Declaration d'un type vecteur */
/* et d'une variable de ce type vecteur */
type arrtype = array of int in
var 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.

```
/* Definitions de type recursives */

/* une liste */
type intlist = {hd: int, tl: intlist} in

/* un arbre */
type tree = {key: int, children: treelist}
and treelist = {treehd: tree, treetl: treelist} in

var 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
var buffer := 'x' in
function readlist() : list =
  function readaint(any: any) : int =
    var i := 0 in
```

