Projet de Compilation (Maîtrise 2000/2001)
$Revision : 1.5$

Roberto Di Cosmo

$Date : 2000/11/1120 : 34 : 06$

Table des matières

Le projet.

Le but de ce projet est de réaliser le front-end d'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.

Dans cette version de l'énoncé, la syntaxe du langage est définie formellement de façon très précise, avec des exemples de programmes.

---

1 Groupes, date de remise des rapports et soutenance du projet

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

Les projets devront être rendus le lundi 15 Janvier 2001 : il faudra remettre un rapport, qui détaillera vos choix pour chacun des phases, et eventuellement les limitations de votre réalisation par rapport à ce qui était demandé ; l'ensemble du code source ne consitute nullement un rapport en soi.
Les soutenances se dérouleront cette même sémaine dans le creneau horaire des TD/TP.

On attend aussi un jeu de tests, et toutes les sources du projet qui doivent compiler correctement avec OCaml version 2.04 ou successives (vous pouvez tester sur le CD de Demo). Cela devra être envoyé par courrier éléctronique sous la forme d'un archive `tar` à `dicosmo@pps.jussieu.fr` et à `buccia@pps.jussieu.fr`.

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 recursive
- définir des types utilisateur, à partir des types de base "string", "int" et "float" à l'aide d'enregistrements et vecteurs
- définir des types locales (avec portée statique)
- définir des types mutuellement recursifs

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 :

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"
```

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

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" , parce-que elles permettent de mettre ensemble plusieurs expressions. On retrouve :
- la concatenation 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 recursives)
- `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 recursives)

Il nous reste a spécifier ce que c'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 réels en format scientifique (comme -3.452E-24)

Enfin, on peut introduire des commentaires, même imbriqués, avec les mêmes conventions qu'en C (i.e. `/*` ouvre et `*/` ferme un commentaire).

2.2 Définition de Ctigre en BNF

Une définition formelles de la syntaxe de Ctigre peut être donnée en utilisant une grammaire en forme BNF comme la suivante, ou les symboles terminaux sont entre `'  '`, alors que les autres symboles sont considérés non terminaux. On ne spécifié pas ident, string-literal, integer-literal et float-literal. On rappelle que 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		
$S \to [S']$	$S \to \epsilon$		$S \to S'$
$S \to S'^*$ ou aussi $S \to \{S'\}$	$S \to \epsilon$		$S \to S'S$
$S \to \alpha_1 \mid \ldots \mid \alpha_n$	$S \to \alpha_1$	$\ldots$	$S \to \alpha_n$

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

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

Important : faite bien attention à distinguer les caractères |, {,}, [,] de la métanotation 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 `]`.

2.3 Grammaire BNF du langage Ctigre

Voici donc la grammaire de CTigre en BNF. Bien entendu, cette grammaire est ambiguë a 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 represente la prémière partie du projet).

```
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 | float-literal | char-literal | 'nil'

type-id  ::=  ident

label  ::=  ident
```

Les symboles non spécifiés (comme ident, integer-literal, string-literal, float-literal) sont à définir par vos soins (à partir des définitions informelles données plus haut).

3 Exemples de programmes Ctigre légaux

Voici quelques exemples de programmes légaux

3.1 Factoriel

Il est possible de déclarer des fonctions récursives.

```
/* Voilà un exemple simple de programme Ctigre: le factoriel */

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

3.2 Vecteurs

Il est possible de déclarer des types utilisateur à partir de constructeurs de types préefinies 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
```

3.3 Types recursifs

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

```
/* Définitions de type recursives */

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

3.4 Un exemple plus complexe : la fusion de deux listes

```
/* Et voici un exemple de programme plus complexe: la fusion de deux listes
   lues sur l'entrée standard */

type any = {any : int} in
type list = {first: int, rest: list} in
let buffer := getchar() in
let readlist() : list =
  let readint(any: any) : int =
    let i := 0 in
    let isdigit(s : string) : int =
        ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
    and skipto() =
         while buffer=" " | buffer="\verb|\|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 := readint(any)
     in if any.any
         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(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("\verb|\|n")
       else (printint(l.first); print(" "); printlist(l.rest))
in
  let list1 := readlist()
  and list2 := (buffer:=getchar(); readlist())
in

  /* CORPS DU PROGRAMME PRINCIPAL */
 printlist(merge(list1,list2))
```

4 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 arithmetiques et logiques sur les types de base, on vous demande de réaliser les fonctions de librairie suivantes :

entrée :

- **print(s)** imprime la chaíne de caractères s
- **printint(i)** imprime l'entier i
- **printfloat(f)** imprime le flottant f

sortie :

- **getchar()** lit un caractère en entrée (opération bloquante)
- **readint()** lit un entier
- **readfloat()** lit un flottant

conversions :

- **ord(c)** le code ASCII du caracère c
- **char(i)** le caractère de code ASCII i

chaînes :

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

5 Phases du projet

Le projet se déroulera en quatre phases assez indépendantes.

5.1 Phase 1 : analyse

Écrivez un analyseur lexicale et un analyseur syntaxiques adaptes à reconnaître un programme Ctigre.

Dans la spécification OcamlYacc, chaque règle de précédence *doit être clairement expliquée*, et tout conflit shift/reduce restant *doit être justifié*.

On n'acceptera pas de spécifications contenant des conflits reduce/reduce.

Soyez particulièrement attentifs à respecter dans votre choix de précédences les conventions que vous pouvez retrouver en Ocaml, par exemple,

```
if true then 3 else 4 *5
```

est à comprendre comme

```
if true then 3 else (4*5)
```

et pas comme

```
(if true then 3 else 4) * 5
```

En cas de doute, testez avec Ocaml d'abord, et reproduisez ensuite la précédence appropriée dans votre grammaire.

5.2 Phase 2 : construction de l'arbre de syntaxe abstraite

Définissez le type `ast` des arbres de syntaxe abstraite de Ctigre, et modifiez votre analyseur syntaxique de telle sorte qu'il puisse retourner un arbre de syntaxe abstraite en cas de reconnaissance réussie.
Dans le type `ast`, il faudra prévoir de préserver une information `positions_source` (une couple d'entier qui disent quels caractères du programme source correspondent a l'objet représente par un noeud de l'`ast`).

Cette information peut être très pratique pour afficher des messages d'erreur raisonnables.

Prévoyez aussi une fonction d'impression (non graphique !) des `ast`, pour que l'on puisse facilement tester le fonctionnement de l'analyseur.

5.3 Phase 3 : analyse sémantique et calcul des attributs

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 dans `positions_source`.

5.3.1 Typage

Dans CTigre l'egalite des types complexes n'est pas structurelle, mais definitionnelle. Autrement dit, si on trouve deux declarations 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 types incompatibles).

Dans CTigre on autorise la definition recursive de type, à condition que toute recursion passe à travers d'un constructeur de type (array ou record).

Donc ceci est illegal :

```
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 specifier leur type, comme dans

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

Un programme CTigre legale permet ces abbreviations seulement si le type de l'identificateur declare peut etre deduit du contexte (comme dans l'exemple), mais pas dans les autres cas, comme celui qui suit (qui n'est pas legal)

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

En effet, `nil` etant une constante qui appartient à tout type enregistrement, on ne peut pas determiner le type de `a` si on ne le specifie 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
```

5.4 Tests

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

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

6 Modularité du projet

Ce projet est clairement séparé en phases distinctes, pour vous permettre, si vous rencontré des difficultés, de ne pas arriver à la fin les mains vides : d'abord l'analyse syntaxique, ensuite la construction de l'arbre de syntaxe abstraite, ensuite l'analyse sémantique .

Cela vous permettra, par exemple, de construire un arbre abstrait pour un programme Ctigre complet même si vous n'arrivez pas à traiter correctement certaines constructions avancées comme les types recursifs : dans ce cas, il faudra émettre un erreur de compilation dans la phase de génération de code ou d'analyse sémantique, mais vous pourrez quand même imprimer l'arbre abstrait du programme que vous ne savez pas compiler.

Aussi, vous pouvez avancer sur le projet en parallèle avec le cours.

Important : prevoyez des options de compilation pour votre compilateur qui permettent de produire une description des étapes intermédiaires. Un bon compilateur devrait au minimum reconnaître les options suivantes[2] :

-ast l'arbre de syntaxe abstraite

-tast l'arbre décoré avec les types

etc.

[2] Le module de librairie `Arg` de Ocaml permet de traiter très facilement les options de la ligne de commande.