Projet Compilation : PreScript

Première partie, cahier des charges

Maîtrise d'Informatique 2002-2003

1 Compilation de votre projet

Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `paulin-miquel.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `prescript`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Calendrier

Le projet est à faire impérativement en binôme. Chacune des trois parties qui le composent est à rendre à votre chargé de TP par e-mail[1] au plus tard à la date indiquée par le calendrier suivant :

Partie 1 Lundi 4/11 à 10h00

Partie 2 Lundi 2/12 à 10h00

Partie 3 Mardi 7/01 à 10h00

2 PreScript

L'objectif du projet est de développer un compilateur d'un mini-langage PreScript vers le langage PostScript qui sert de langage graphique d'entrée à de nombreuses imprimantes et outils d'affichage. La langage PreScript est un langage avec typage explicite, qui permet de manipuler en particulier des tableaux et des types étiquetés. Un exemple de programme PreScript est :

```
(* Type des points repérés par deux flottants *)
type point = {abs : float; ord : float}
(* Définition de deux points particuliers *)
var p0 = {abs = 1; ord = 0}
var p1 = {abs = 0; ord = 1}
(* Utilisation de la fonction primitive sqrt *)
var p2 = {abs = sqrt(2)/2; ord = sqrt(2)/2}
(* Tableau formé de deux points *)
var base = [p0;p1]
(* fonction de symétrie par rapport à l'axe des y *)
fun symy (p:point) : point = {abs=-p.abs;ord=p.ord}
(* Fonction effectuant le produit scalaire de deux vecteurs de flottants *)
fun scalaire (r1:float array,r2:float array) : float =
    var n = length (r1) in
    var r = 0. in
    for i = 0 to n-1 do r:=r+r1.(i)*r2.(i) done;
    r
```

3 Conventions lexicales de PreScript

3.1 Blancs

Les caractères espace, retour chariot et tabulation sont considérés comme blancs. Ils sont pris en compte uniquement lors de l'analyse lexicale en tant que séparateurs et sont ignorés par la suite.

[1] `frederic.de.jaeger@ens.fr`, `miquel@lri.fr`, `paulin@lri.fr`, `clement.renard@inria.fr`, `signoles@lri.fr`

3.2 Commentaires

Les débuts de commentaires sont marqués par le lexème '`(*`' et les fins de commentaires par le lexème '`*)`'. Les parties d'un programme qui sont commentées doivent être traitées de la même façon que les blancs par l'analyseur lexical.

Il est impératif de pouvoir commenter un bout de programme facilement. Plus précisément, si on prend un programme correct, on devra pouvoir commenter toute suite de lexèmes en ajoutant un début de commentaire entre deux lexèmes ainsi qu'une fin de commentaire entre deux lexèmes, plus loin dans le programme. Lorsqu'on fait cela, toute la zone comprise entre ces deux marques de commentaires est considérée comme étant commentée.[2]

Si un programme est lexicalement correct, l'ajout d'un début de commentaire sans ajout d'une fin de commentaire doit le rendre incorrect.

Afin d'avoir la plus grande liberté possible pour écrire du texte entre commentaires, on essaiera d'autoriser d'écrire le plus possible de choses entre une marque de début de commentaire et une marque de fin de commentaire tant que cela ne contrevient pas aux règles précédentes sur les commentaires.

3.3 Mots clés

Les mots clés sont les suivants : `array`, `begin`, `div`, `do`, `done`, `else`, `end`, `false`, `for`, `fun`, `if`, `in`, `mod`, `not`, `step`, `then`, `to`, `true`, `type`, `var`, `while`

3.4 Identificateurs (*id*)

Les identificateurs sont constitués d'une suite non vide de caractères dont le premier est alphabétique et dont tous les autres sont alphanumériques ou '`_`'. Leur longueur est arbitraire. Les mots clés de PreScript ne peuvent être acceptés comme identificateurs.

3.5 Constantes (*const*)

Les constantes sont les constantes entières, flottantes, caractères, chaînes, booléennes et une constante particulière appelée unit.

- Une constante entière est une suite non vide de chiffres.
- Une constante flottante est une suite non vide de chiffres, suivie d'un point, d'une suite éventuellement vide de chiffres, et éventuellement du caractère '`e`' ou du caractère '`E`' suivis par une suite non vide de chiffres éventuellement précédée d'un signe ('`+`' ou '`-`').
- Une constante caractère se commence et se termine par le caractère '`'`'. Elle contient soit un caractère autre que '\' et '`'`', soit une séquence d'échappement. Une séquence d'échappement est '\\', '\`n`', '\`t`' ou '\`'`'. Ces séquences d'échappement représenteront respectivement les caractères '\', retour chariot, tabulation et '`'`'.
- Une constante chaîne se commence et se termine par le caractère '`"`'. Elle contient une suite de séquences, une séquence étant soit un caractère autre que '\' et '`"`', soit une séquence d'échappement. Une séquence d'échappement est '\\', '\`n`', '\`t`' ou '\`"`'. Ces séquences d'échappement représenteront respectivement les caractères '\', retour chariot, tabulation et '`"`'.[3]
- Les constantes booléennes sont `true` et `false`.
- La constante de type `unit` est notée `()`.

4 Syntaxe de PreScript

La syntaxe que votre compilateur doit accepter est donnée par la grammaire suivante, de symbole de départ *prog* :

$$
\begin{array}{lcl}
\textit{prog} & ::= & \epsilon \\
 & | & \textit{prog decl} \\
 & & \\
\textit{decl} & ::= & \texttt{type}\ \textit{typid} = \textit{type} \\
 & | & \texttt{var}\ \textit{id} = \textit{expr} \\
 & | & \texttt{fun}\ \textit{id}\ \texttt{(}\ \textit{opt\_params}\ \texttt{)}\ \texttt{:}\ \textit{type} = \textit{expr}
\end{array}
$$

[2]Notez que le langage C ne vérifie pas cette propriété car les commentaires ne peuvent être imbriqués : pour commenter une partie de code contenant des commentaires, il ne suffit donc pas d'insérer un début de commentaire au début de cette partie et une fin de commentaire à la fin.

[3]Attention : les caractères simples et les séquences d'échappement ne sont pas tout-à-fait les mêmes dans les chaînes et dans les caractères. L'apostrophe s'écrit '\`'`' dans une constante caractère mais '`'`' dans une chaîne, tandis que les guillemets s'écrivent '`"`' dans une constante caractère mais '\`"`' dans une chaîne.

```
opt_params ::= ε
             | params

params ::= param
         | params , param

param ::= id : type

type ::= typid
       | type array
       | { field_decls }

field_decls ::= field_decl
              | field_decls ; field_decl

field_decl ::= label : type

expr ::= const
       | id
       | id ( opt_args )
       | unop expr
       | expr binop expr
       | ( expr )
       | [ opt_expr_list ]
       | { field_defs }
       | begin expr end
       | var id = expr in expr
       | if expr then expr
       | if expr then expr else expr
       | for id = expr to expr do expr done
       | for id = expr to expr step expr do expr done
       | while expr do expr done
       | expr . label
       | expr .( expr )
       | expr .[ expr ]
       | lval := expr
       | expr ; expr

opt_expr_list ::= ε
                | expr_list

expr_list ::= expr
            | expr_list ; expr

opt_args ::= ε
           | args

args ::= expr
       | args , expr

unop ::= - | not

binop ::= + | - | * | / | div | mod
        | = | <> | < | > | <= | >=
        | && | ||

lval ::= id
       | lval . label
       | lval .( expr )
       | lval .[ expr ]

field_defs ::= field_def
             | field_defs ; field_def

field_def ::= label = expr
```

Types Les identificateurs de type (*typid*) et les labels (*label*) sont des identificateurs standards (*id*), à ceci près qu'ils vivent dans des espaces de nommage différents. On pourra donc utiliser un même identificateur comme valeur, comme type et comme label simultanément. Les types de base `unit`, `bool`, `int`, `float`, `char`, `string` et `path` sont simplement des identificateurs de types prédéfinis et non des mots clés.

Fonctions Les opérateurs arithmétiques (`+`, `-`, `*`, `/`, `div` et `mod`) sont munis des priorités habituelles. Les opérateurs de comparaison (`=`, `<>`, `<`, `>`, `<=` et `>=`) doivent avoir une priorité plus faible, et les opérateurs logiques une priorité encore plus faible, avec `not` prioritaire sur `&&`, lui-même prioritaire sur `||`. Le `if` ... `then` ... `else` est prioritaire sur la séquence *expr* `;` *expr*, elle-même prioritaire sur le `var` ... `in`. Le langage comportera un certain nombre de fonctions prédéfinies pour compléter les opérations arithmétiques et introduire des primitives graphiques.

Parenthésage Les parenthèses `(` ... `)` et le `begin` ... `end` ne servent qu'à grouper les termes. Une expression de la forme `(` *expr* `)` ou `begin` *expr* `end` doit donc être analysée de la même façon que l'expression *expr*.

Tableaux & chaînes L'expression $[e_0; \ldots; e_n]$ représente un tableau de longueur $n+1$ dont l'élément d'indice i est la valeur de e_i. Si t est un tableau et e une expression qui s'évalue en un entier i alors $t.(e)$ représente la valeur du tableau t à l'indice i. Les valeurs des tableaux peuvent être modifiées en place par l'instruction $t.(e_1)$ `:=`e_2.

Si t est une chaîne de caractères et e une expression qui s'évalue en un entier i alors $t.[e]$ représente le i-ème caractère de t. Les caractères d'une chaîne peuvent être modifiés en place par l'instruction $t.[e_1]$ `:=`e_2.

5 Invocation du compilateur

La commande `prescript` [*options*] *fichier*$_1$ ...*fichier*$_n$ compile les fichiers *fichier*$_1$, ..., *fichier*$_n$. La valeur de retour de la commande est

- 0 si tout s'est bien passé ;
- 1 si le fichier donné en entrée est incorrect ;
- 2 si vous détectez une erreur dans votre compilateur (par exemple, vous arrivez sur un cas qui n'aurait pas dû être possible).

Notez que le code de retour 2 est le code de sortie d'un programme Caml qui lance une exception et ne la rattrape pas. Les options possibles doivent comprendre au moins :

-help : affiche un message décrivant la façon d'invoquer le compilateur.

-about : affiche les noms des auteurs, et éventuellement le numéro de version du compilateur.

-no-parse : s'arrête avant l'analyse syntaxique. En d'autres termes, effectue uniquement l'analyse lexicale.

-no-sem : s'arrête avant l'analyse sémantique. Autrement dit, effectue uniquement l'analyse lexicale et l'analyse syntaxique.

Dans la première partie, `prescript` échouera si on ne se restreint pas à faire l'analyse lexicale ou à l'analyse syntaxique et devra calculer l'arbre de syntaxe abstraite . `prescript` se contentera de faire selon les cas l'analyse lexicale et syntaxique de chacun des fichiers qui lui sont donnés et de calculer l'arbre de syntaxe abstraite de ces fichiers ou simplement l'analyse lexicale (en mode `-no-parse`).

Vous pouvez de façon optionnelle ajouter une option **-debug** qui fera afficher diverses informations de déverminage à votre compilateur.

En cas d'erreur lexicale, `prescript` doit afficher un message sur la sortie standard d'erreur indiquant la position dans le fichier original de l'erreur en respectant la syntaxe utilisée dans l'exemple suivant :

```
File "toto.ml", line 10, characters 24-29 : Lexical error.
```

De même, en cas d'erreur syntaxique on écrira :

```
File "titi.ml", line 277, characters 10-96 : Syntax error.
```