Compilation
Troisième partie, cahier des charges

À rendre le 7 janvier 2003 avant 10h

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*. 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.

Votre rapport devra être rendu au moment de la soutenance. Outre ce qui est spécifique à cette troisième partie, votre rapport devra contenir, de manière synthétique (une page), un aperçu global des fonctionnalités de votre compilateur et des difficultés rencontrées.

2 Éléments de programmation en PostScript

2.1 Machine à pile

PostScript [1, 2] est un langage dont l'exécution repose sur une machine à pile (appelée *pile des opérandes*). Chaque commande PostScript prend ses arguments sur la pile, les dépile, et place le résultat de l'opération au sommet de la pile. Par exemple la séquence

```
5 3 sub
```

empile d'abord les entiers 5 et 3 dans cet ordre. L'exécution de la commande `sub` dépile ces deux éléments et rempile le résultat de la soustraction (ici 2) sans changer le contenu du reste de la pile.

2.2 Manipulation des noms

Les définitions de noms s'effectuent de la manière suivante :

```
/foo (Hello) def       % place la chaîne (Hello) dans foo
/bar 1 2 add 4 mul def  % place l'entier 12 dans bar
```

(Le caractère '/' indique qu'on réfère au nom plutôt qu'à la valeur qu'il représente.) On peut utiliser ensuite ces noms à n'importe quel endroit du programme :

```
bar 10 sub   % place l'entier 2 sur le sommet de la pile
```

2.3 Types de données

En PostScript, le typage est dynamique, ce qui signifie que le type d'un objet n'est consulté qu'au moment de l'exécution d'une primitive l'utilisant. Les types dont vous aurez besoin sont :

- Le type `nulltype`, qui ne comporte que la valeur `null`.
- Le type `booleantype`, qui ne comporte que les valeurs `true` et `false`.
- Le type `integertype` des entiers.
- Le type `realtype` des flottants. Par extension, un entier peut être toujours utilisé là où un flottant est attendu, l'interpréteur faisant la conversion automatiquement.
- Le type `stringtype` des chaînes de caractères, qui s'écrivent entre parenthèses. Par exemple : `(Hello !)`.
- Le type `arraytype` des tableaux, qui peuvent contenir des valeurs de n'importe quel type. Syntaxiquement, les tableaux sont délimités par des crochets ('`[`' et '`]`'). Il peut y avoir des commandes PostScript arbitraires entre les crochets délimitant un tableau, lesquelles sont executées de la gauche vers la droite avant la construction de celui-ci. Ainsi, la séquence

  ```
  [ 1 2 add (Hello) [ 3.2 4 mul ] ]
  ```

  construit le même tableau que la séquence

  ```
  [ 3 (Hello) [ 12.8 ] ]
  ```

  Il existe également une variante des tableaux : les *procédures* (ou *tableaux exécutables*) sur lesquelles nous reviendrons au paragraphe 2.5.
- Le type `nametype` des noms, qui peuvent être sous forme littérale (*i.e.* précédée par un `/`) ou exécutable (*i.e.* sans le `/`).
- Le type `dicttype` des dictionnaires (*cf* paragraphe 2.4).

2.4 Dictionnaires

Les associations entre noms et valeurs sont stockées en PostScript dans des *dictionnaires*. Le langage dispose d'une pile supplémentaire, appelée *pile des dictionnaires*, qui est consultée à chaque définition de nom et à chaque appel de sa valeur. Plus précisément :

- La définition d'un nom x s'effectue toujours dans le dictionnaire d au sommet de la pile des dictionnaires. Si le nom x est déjà défini dans d, l'ancienne définition est écrasée. Sinon, une nouvelle association est ajoutée à d, qui masque les associations éventuelles du nom x dans les dictionnaires suivants.

– L'accès à la valeur associée à un nom s'effectue en parcourant la pile des dictionnaires à partir de son sommet, jusqu'à rencontrer une association de ce nom dans un des dictionnaires de la pile.

On peut créer un dictionnaire de taille n avec la commande n `dict`, et l'ajouter au sommet de la pile des dictionnaires avec la commande `begin`. Symétriquement, on peut dépiler le dictionnaire au sommet de la pile des dictionnaires avec la commande `end`. Par exemple :

```
/x 3 def
2 dict begin     % ajoute un nouveau dictionnaire de
                 % taille 2 au sommet de la pile des
                 % dictionnaires
  /x (toto) def
  /y true def
  x ==             % affiche: (toto)
  y ==             % affiche: true
end              % dépile le dictionnaire
x ==             % affiche: 3
y ==             % déclenche une erreur (y non défini)
```

2.5 Procédures

En PostScript, une procédure est une séquence de commandes comprise entre deux accolades ('`{`' et '`}`'). Les commandes ne sont pas exécutées au moment de la création de la procédure, mais à chaque exécution de celle-ci, typiquement au travers de l'utilisation d'un nom. Par exemple :

```
/i 0 def                          % initialisation du compteur
/incr { /i i 1 add def } def      % procédure d'incrémentation
i == incr i == incr i ==          % affiche: 0, 1 puis 2
```

2.6 Principales commandes

Nous donnons ici un rapide aperçu des principales commandes PostScript potentiellement nécessaires dans le cadre de ce projet. (À vous de choisir lesquelles !)

Manipulation de la pile Les principales commandes PostScript qui permettent de manipuler la pile sont les suivantes :

n `index`, `dup`, `exch`, `pop`, `clear`, `pstack`

La commande n `index` place au sommet de la pile l'élément situé au rang n dans la pile (l'élément situé juste avant l'indice n étant considéré comme l'élément de rang 0). La commande `dup` duplique l'élément au sommet de la pile et la commande `exch` échange les deux premiers éléments au sommet de la pile. La commande `pop` dépile le dernier élément empilé, tandis que la commande `clear` efface toute la pile. Enfin, la commande `pstack` affiche le contenu de la pile sans modifier celle-ci ; elle est le plus souvent utilisée à des fins de déverminage.

Opérateurs arithmétiques et logiques Les opérateurs unaires et binaires s'utilisent de la manière suivante :

> *arg* ***unop***
> arg_1 arg_2 ***binop***

Les opérateurs unaires et binaires PostScript dont vous aurez besoin sont : `neg`, `not`, `add`, `sub`, `mul`, `div`, `idiv`, `mod`, `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `and` et `or`.

Conditionnelle Il existe en PostScript deux formes de conditionnelle :

> *cond* $proc_1$ `if`
> *cond* $proc_1$ $proc_2$ `ifelse`

Ces commandes exécutent la procédure $proc_1$ si *cond* est égale à `true`. La première ne fait rien si *cond* est égale à `false`, tandis que la seconde exécute la procédure $proc_2$ dans ce cas.

Boucles Il existe en PostScript trois types de boucles :

> *deb inc fin proc* `for`
> *num proc* `repeat`
> *proc* `loop`

La commande `for` itère la procédure *proc* sur l'ensemble des valeurs allant de *deb* à *fin* en ajoutant à chaque pas la valeur *inc*. À chaque itération, la valeur courante du compteur est placée sur le sommet de la pile juste avant l'appel de la procédure *proc*. La commande `repeat` exécute *num* fois la procédure *proc*, tandis que la commande `loop` répète l'exécution de cette procédure à l'infini. On peut toujours interrompre l'exécution d'une boucle à l'aide de la commande `exit`.

Manipulation des chaînes Les chaînes de caractères sont manipulées en PostScript à travers les commandes suivantes :

> *num* `string`
> *str idx* `get`
> *str idx chr* `put`

La commande `string` crée une nouvelle chaîne de taille *num*, dont le contenu est indéfini, et qui est placée au sommet de la pile. La commande `get` accède au caractère de rang *idx* dans *str* et place le code ASCII correspondant au sommet de la pile. Enfin, la commande *put* affecte au rang *idx* de la chaîne *str* le caractère de code ASCII *chr*.

Manipulation des tableaux Les tableaux sont manipulés de la même manière que les chaînes de caractères à travers les commandes suivantes :

num `array`
tbl idx `get`
tbl idx obj `put`

La commande *num* `array` retourne un tableau de longueur *num* dont tous les éléments sont initialisés avec la valeur `null`. On rappelle également qu'il est possible de stocker des objets de type arbitraire dans les tableaux.

Manipulation des dictionnaires Les dictionnaires sont manipulés au moyen des commandes suivantes.

num `dict`
dic /nom `get`
dic /nom obj `put`
dic `begin`
`end`

La commande *num* `dict` construit un nouveau dictionnaire de taille *num*, vide de toute association. La commande `get` (resp. `put`) permet d'accéder à (resp. d'affecter) la valeur associée à un nom dans un dictionnaire. Enfin, la commande *dic* `begin` empile le dictionnaire *dic* au sommet de la pile des dictionnaires tandis que la commande `end` dépile le dernier dictionnaire empilé.

3 Représentation des données

Les structures de données du langage PreScript seront représentées en PostScript de la manière suivante :

3.1 Types de base

Les booléens, les entiers, les flottants et les chaînes de caractères du langage PreScript seront représentés respectivement par les booléens, les entiers, les flottants et les chaînes de caractères du langage PostScript.

La valeur `()` de type `unit` sera représentée par la constante `null`.

Comme il n'existe pas de type caractère en PostScript, les caractères du langage PreScript (type `char`) seront représentés par leur code ASCII dans le programme compilé.

Au moment de la production du code, on veillera à respecter les conventions lexicales de PostScript, qui ne sont pas toujours les mêmes que celles de PreScript. Comme les chaînes de caractères sont délimitées par des parenthèses ('`(`' et '`)`') en PostScript, et non par des guillemets droits ('`"`'), les séquences d'échappement ne sont pas les mêmes, ainsi que l'illustrent les exemples suivants :

PRESCRIPT	POSTSCRIPT
`"Hello!\n"`	`(Hello!\n)`
`"(Hello!)"`	`(\(Hello!\))`
`"\"Hello!\" dit-il"`	`("Hello!" dit-il)`

3.2 Tableaux et enregistrements

Les tableaux et les enregistrements du langage PRESCRIPT seront représentés tous deux par des tableaux. Par convention, les champs des enregistrements seront disposés suivant l'ordre lexicographique des labels correspondants. Ainsi, l'enregistrement

```
{ re = 1.0 ; im = 0.0 }
```

sera représenté par le tableau suivant en POSTSCRIPT :

```
[ 0.0 1.0 ]
```

(Cette convention a l'avantage de rendre la représentation des enregistrements insensible à l'ordre de déclaration des champs.)

4 Compilation

Il y a plusieurs manières de compiler le langage PRESCRIPT. Nous vous proposons ici un schéma de compilation simple, mais rien ne vous empêche d'en suivre un autre, du moment qu'il est correct.

Dans ce qui suit, on notera $[\![e]\!]$, $[\![d]\!]$ ou $[\![p]\!]$ le code produit par la compilation de l'expression e, de la déclaration d ou du programme p.

4.1 Expressions

Lors de la compilation d'une expression e, on prendra garde à respecter l'invariant selon lequel l'exécution du code $[\![e]\!]$ place au sommet de la pile la valeur calculée par e sans modifier le reste de la pile. En particulier, on fera attention à utiliser correctement les commandes `if`, `for`, `repeat`, `loop`, `put`, etc. qui, en POSTSCRIPT, n'ont pas été conçues pour respecter cet invariant.

Les variables locales seront représentées par des noms en POSTSCRIPT (on prendra garde à ne jamais utiliser un nom qui entrerait en conflit avec un nom de primitive POSTSCRIPT). Par exemple, l'expression

```
var x = 3 + y in x * x
```

pourra être compilée par la suite de commandes :

```
/_x_19 3 _y_18 add def _x_19 _x_19 mul
```

Enfin, l'application $f(e_1, \ldots, e_n)$ sera compilée comme l'application de la procédure représentant f (*cf* paragraphe 4.3) aux n premiers éléments de la pile produits par l'exécution successive des codes de $e_1, \ldots, e_n$, c'est-à-dire :

$$[\![f(e_1, \ldots, e_n)]\!] \;=\; [\![e_1]\!] \;\cdots\; [\![e_n]\!] \;\mathit{nom\_f}$$

4.2 Enregistrements

La principale difficulté du schéma de compilation des enregistrements vers les tableaux réside dans le fait qu'un même nom de label peut être associé à des indices différents suivant le type de l'enregistrement considéré. Dans l'exemple suivant

```
var x = { a = 'A' ; b = 18 } in
var y = { c = false ; b = 275 } in
x.b + y.b
```

le label `b` n'aura pas la même position dans le tableau associé à `x` (ici, la position 1) ou dans celui associé à `y` (ici, la position 0), ainsi que l'illustre le code compilé correspondant :

```
/_x_1 [ 65 18 ] def
/_y_2 [ 275 false ] def
_x_1 1 get _y_2 0 get add
```

Pour résoudre cette difficulté, nous vous suggérons de modifier le type des labels dans votre arbre de syntaxe abstraite de la manière suivante :

```
type label = { nom : string; mutable indice : int }
```

(Ces modifications devront être répercutées dans toutes les parties du code où c'est nécessaire, ce qui ne devrait pas occasionner beaucoup de changements.)

Avec cette nouvelle représentation, chaque occurrence d'un label dans l'arbre de syntaxe abstraite comporte à présent une référence `indice`. Cette dernière, initialisée avec une valeur arbitraire lors de l'analyse syntaxique, recevra au moment du typage la position du champ désigné par ce label dans l'enregistrement correspondant. Cette position sera ensuite utilisée lors de la production de code comme un indice de tableau.

4.3 Déclarations

Les déclarations de types disparaissent à la compilation, et les déclarations de variables globales s'effectuent de la même manière que pour les variables locales. Enfin, les déclarations de fonctions seront compilées comme des définitions de procédures selon le schéma suivant :

$[\![\texttt{fun}\ f\,(x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ e]\!] \quad =$

$\quad /nom\_f\,\texttt{\{}$
$\quad\quad p\ \texttt{dict begin}$
$\quad\quad\quad /nom\_x_n\ \texttt{exch def}$
$\quad\quad\quad\quad \vdots$
$\quad\quad\quad /nom\_x_1\ \texttt{exch def}$
$\quad\quad\quad [\![e]\!]$
$\quad\quad \texttt{end}$
$\quad \texttt{\}}\ \texttt{def}$

Dans le code ci-dessus, *nom_y* désigne le nom PostScript correspondant à la variable y dans le langage PreScript. Par ailleurs, l'entier p désigne ici la taille du dictionnaire local qui sera créé à chaque appel de procédure. On veillera donc à calculer une valeur adéquate pour ce paramètre avant de compiler le code. (Exercice préliminaire indispensable : comprendre la signification du code ci-dessus !)

4.4 Primitives

Le langage PreScript comporte un certain nombre de primitives permettant d'accéder aux fonctions graphiques du langage. Ces fonctions sont définies dans un fichier `prelude.ps` que vous trouverez dans le répertoire `~signoles/tp_compil`, et qui devra être recopié en tête du code compilé. La liste complète des primitives vous sera donnée ultérieurement.

5 Invocation du compilateur

La commande `prescript` [*options*] *fichier* compile le fichier *fichier*. 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).

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 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.

-no-code : s'arrête avant la production de code ; autrement dit, effectue uniquement les analyses lexicale, syntaxique et sémantique.

-o : pour nommer le fichier de sortie (par défaut il devra s'appeler *a.ps*).

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

Références

[1] PostScript Language – Tutorial and Cookbook, 3e édition. Adobe Systems Incorporated. Addison-Wesley (ISBN-0-201-10179-3), mai 1986.

[2] PostScript Language – Reference manual, 2e édition. Adobe Systems Incorporated. Addison-Wesley (ISBN-0-201-10174-2), décembre 1985.