UNIVERSITÉ DE NANTES

Master 1 Alma

Projet de TP de compilation

Conception d'un compilateur d'images SVG

Étudiant :
Vincent Raveneau

Intervenant :
Benoît Guédas

10 Avril 2014

Sommaire

Présentation du projet

Afin de mettre en pratique les différents éléments étudiés lors du cours de compilation et des TPs associés, nous avons eu pour tâche de réaliser un compilateur pour un langage permettant de générer des images au format SVG. Bien que les principaux éléments du langage en question nous aient été imposés par le sujet, leur implémentation restait libre et pouvait être réalisée de différentes façons.

Le compilateur devait être codé en Ocaml, à l'aide d'ocamllex et d'ocamlyacc, et sa réalisation devait être décomposées en différentes étapes successives. Bien que l'ordre et le contenu de ces étapes soient entièrement libres, cette stratégie avait pour but de nous permettre de nous concentrer sur une tâche à la fois. De plus, celà garantissait que lors du déroulement d'une étape, tous les aspects du langage définis dans les étapes précédantes étaient pleinement fonctionnels et non partiellement implémentés.

Ce rapport a donc pour but de présenter le travail que j'ai effectué dans le cadre de ce projet. Une présentation du langage implémenté sera d'abord effectuée, puis chaque étape dans la création du compilateur sera décrite en détail, afin d'illustrer la démarche itérative mise en place.

Présentation du langage implémenté

2.1 Caractéristiques

Le langage implémenté doit permettre à l'utilisateur de décrire une image au format SVG. Pour celà, il dispose d'un langage de type impératif, dont la syntaxe est similaire au C.

[Parler du typage fort ou non, explicite ou non]

2.2 Syntaxe

[Exemple de code et description des éléments clefs (présentation des erreurs à ne pas faire, genre $0. != 0.0$)]

2.3 Éléments non implémentés

Bien que demandés par le sujet, certains éléments principaux du langage n'ont pas été implémentés faute de temps. C'est le cas notamment de la gestion des erreurs, et des structures de contrôle telles que les boucles et les conditionnelles (de type `if (condition) then (instructions) else (instructions)`).

Étape 1 : Les types primitifs

Lors de cette étape initiale, l'objectif était de permettre au compilateur de détecter les types primitifs du langage, qui sont les suivants :

- Les entiers : une suite de chiffres allant de 0 à 9 ;
- Les flottants : un entier suivi d'un point et d'un autre entier ;
- Les chaînes de caractères : une suite de caractères encadrée par des guillemets droits (`"`) ;
- Les booléens : deux valeurs possibles, `true` et `false` ;
- Les couleurs : six couleurs possibles, `red`, `blue`, `yellow`, `green`, `black` et `white`.

À cette étape du projet, lorsqu'un élément correspondant à un de ces types est identifié, son type et sa valeur sont affichés sur la sortie standard mais aucune action n'est effectuée dessus.

[Parler des choix, de l'implémentation]

Étape 2 : Opérations sur les types primitifs

Une fois les types primitifs implémentés, les opérateurs usuels correspondant ont été implémentés. Ceux-ci peuvent être de trois sortes, selon les types auxquels ils s'appliquent.

Tout d'abord, les opérateurs arithmétiques ont été mis en place. Ceux-ci sont l'addition (`+`), la soustraction (`-`), la multiplication (`*`), la division (`/`) et le modulo (`%`). Ces opérateurs s'appliquent aux entiers et aux flottants, à l'exception du modulo qui ne s'applique par définition qu'aux entiers. Il est cependant important de noter que les deux opérandes associées à un opérateur doivent être de même type (par exemple, l'addition d'un entier avec un flottant est impossible).

Ensuite, les opérateurs de comparaison ont été implémentés. Ceux-ci s'appliquent également aux entiers et au flottants, et les opérandes d'un opérateur doivent également être de même type. On trouve les opérateurs d'infériorité stricte (`<`), de supériorité stricte (`>`), d'infériorité (`<=`) et de supériorité (`=>`).

Enfin, les opérateurs booléens de composition ont été mis en place. Ceux-ci sont le ET logique (`and`) et le OU logique (`or`), et s'appliquent aux booléens comme leur nom l'indique. Le OU est ici non-exclusif, ce qui signifie que l'expression `vrai ou vrai` a pour valeur `vrai`.

La prise en compte des parenthèses afin de définir les priorités dans le calcul des expressions a également été mise en place pour les trois catégories d'opérateurs décrites ci-dessus.

Étape 3 :

- détection des types complexes basiques (sans champs optionnels) - cercle - rectangle - point (pour éviter le problème de nommage dans les records, et pour l'utilité) - ligne - texte - image

Étape 4 :

- détection des types complexes plus complexes (avec champs optionnels) - définition des constructeurs - uniquement constructeurs basiques pour l'instant (param oblig, param oblig + optionnels) - plus de constructeurs rajoutés à la fin si possible (params variés, (x,y)->point , ...)

Étape 5 :

- détection des variables - détection des affectations - enregistrement des variables en mémoire (sans test) - enregistrement des variables en mémoire (avec test) - récupération de la valeur des variables pour réutilisation (- implémentation de la structure des instructions : fin par un ;)

—— production d'un AST entre l'analyse syntaxique et sémantique

- modif de la façon dont les variables sont faites - draw

—— pas respect stricte du fonctionnel (beaucoup de mutable)

- progression du draw (modification de la détection des strings pour en retirer les "")

- prise en charge du dessin de toutes les formes de base (ligne, rectangle, cercle, texte)

—— début du travail sur les structures de controle