UNIVERSITÉ DE STRASBOURG

Master Informatique
Compilation
2016 / 2017

Projet : Compilateur pour le langage TexSci

Le but du projet est de réaliser un compilateur pour le langage TexSci, depuis un fichier LaTeX jusqu'à un code exécutable MIPS.

1 Résumé du projet

LaTeX (prononcer « latek »), est un langage permettant de composer un document. Il est très utilisé notamment pour écrire des documents scientifiques et des articles de recherche [1]. De tels documents peuvent parfois contenir la description d'un algorithme. Pour cela, il existe un environnement appelé *Algorithm2e* qui étend LaTeX en fournissant des commandes plus spécifiques pour écrire des algorithmes. L'objectif de ce projet est d'implémenter un compilateur qui extrait les algorithmes d'un document LaTeX et produit du code exécutable MIPS qui réalise ces algorithmes, c'est à dire un code MIPS dont l'exécution suit les algorithmes qui sont décrits.

Le langage TexSci est un *domain specific language* (ou DSL) basé sur un sous-ensemble de l'environnement *Algorithm2e*. TexSci a pour but de simplifier l'écriture de programmes pour les scientifiques qui utilisent LaTeX pour décrire leurs travaux et leurs algorithmes. Le but de TexSci est de permettre d'obtenir du code exécutable directement à partir d'un document LaTeX qui décrit des algorithmes sans nécessiter l'utilisation d'un autre langage de programmation.

Un premier exemple d'algorithme TexSci est présenté en figure 1. Cette figure montre un algorithme de calcul de la factorielle sous sa forme TexSci en figure 1(a) et le rendu obtenu dans le document produit par LaTeX en figure 1(b). Il s'agira dans ce projet d'écrire un compilateur permettant de passer de la forme en figure 1(a) à un code assembleur MIPS exécutable.

2 Présentation du langage source

2.1 Principe des commandes LaTeX

Le « programme source » est un document texte écrit en LaTeX (contenu dans un fichier dont l'extension est .tex), mais on ne prendra en compte qu'une partie seulement de ce document : la partie qui décrit les algorithmes au format TexSci. Tout le reste sera considéré comme des commentaires. La description suivante de la syntaxe du langage source reste assez informelle : nous laissons au concepteur du compilateur la liberté de choisir les règles de grammaire à condition que ces règles définissent bien un langage « TexSci » c'est à dire un sous-ensemble du langage LaTeX étendu avec les commandes de l'environnement *Algorithm2e*.

Les commandes LaTeX commencent toutes par une barre oblique inversée (ou antislash) `\` (par exemple `\title{Projet de compilation}`) et sont éventuellement immédiatement suivies d'arguments entre accolades. Celles qui nous intéressent particulièrement ici sont celles situées entre `\begin{texsci}` et `\end{texsci}`. La figure 1(a) contient un exemple d'algorithme écrit en LaTeX.

Pour que votre « programme source » puisse inclure des algorithmes TexSci et être traité par LaTeX, il devra contenir les commandes listées en figure 2 au début du fichier .tex, avant la commande `\begin{document}` (nous vous invitons à recopier ce texte depuis l'un des fichiers de test).

1. Ce sujet a été realisé avec LaTeX.

```
\begin{texsci}{factorial}
\Input{$n \in \Integer$}
\Output{$accu \in \Integer$}
\Global{$\emptyset$}
\Local{$\emptyset$}
\BlankLine
  $accu \leftarrow 1$\;
  \While{$n \neq 0$} {
    $accu \leftarrow n \times accu$\;
    $n \leftarrow n - 1$\;
  }
\end{texsci}
```

Algorithm 1: factorial

Input : $n \in \mathbb{Z}$
Output : $accu \in \mathbb{Z}$
Global : $\emptyset$
Local : $\emptyset$

$accu \leftarrow 1$;
while $n \neq 0$ **do**
  $accu \leftarrow n \times accu$;
  $n \leftarrow n - 1$;

```
\begin{texsci}{main}
\Local{$f \in \Integer$}
\BlankLine
  \tcc{Powered by TexSci}
  $f \leftarrow \mbox{factorial($7$)}$\;
  $\mbox{printInt($f$)}$\;
\end{texsci}
```

Algorithm 2: main

Local : $f \in \mathbb{Z}$

/* Powered by TexSci */
$f \leftarrow$ factorial(7);
printInt(f);

(a) Code source TexSci (b) Rendu obtenu avec LaTeX

FIGURE 1 – Exemple de code source TexSci d'un algorithme de calcul de la factorielle

2.2 Éléments syntaxiques des algorithmes TexSci

Dans ce qui suit, les symboles `$` font partie de la syntaxe, mais pas les mots entre `<` et `>`.

La description d'un algorithme commence par `\begin{texsci}{<nom d'algorithme>}` et termine par `\end{texsci}`. Le point d'entrée du programme est l'algorithme ayant pour nom `main`. Les algorithmes peuvent inclure des commentaires sous la forme `\tcc{<commentaire>}`.

2.2.1 Zone des déclarations

Au début de la description d'un algorithme se trouve une zone pour déclarer les constantes et les variables utilisées dans l'algorithme. Cette zone se termine par `\BlankLine`. Les commandes dans cette zone sont :

— `\Constant{$<suite de déclarations>$}` pour déclarer des constantes,
— `\Input{$<suite de déclarations>$}` pour déclarer les variables en entrée (c'est à dire les arguments de l'algorithme),
— `\Output{$<déclaration>$}` pour déclarer la variable en sortie (c'est à dire le résultat de l'algorithme),
— `\Global{$<suite de déclarations>$}` pour déclarer les variables globales à tous les algorithmes du document,
— `\Local{$<suite de déclarations>$}` pour déclarer les variables locales à l'algorithme.

```
% --- START - HEADER TO SUPPORT TexSci ALGORITHMS IN LaTeX DOCUMENTS
\usepackage[vlined,ruled]{algorithm2e}
\usepackage{amssymb}
\usepackage{amsmath,alltt}
\newenvironment{texsci}[1]{%
\vspace{1cm}
\renewcommand{\algorithmcfname}{Algorithm}
\begin{algorithm}[H]
\label{#1}
\caption{#1}
\SetKwInOut{Constant}{Constant}
\SetKwInOut{Input}{Input}
\SetKwInOut{Output}{Output}
\SetKwInOut{Global}{Global}
\SetKwInOut{Local}{Local}
}{%
\end{algorithm}
\vspace{1cm}
}
\newcommand{\true}{\mbox{\it true}}
\newcommand{\false}{\mbox{\it false}}
\newcommand{\Boolean}{\{\true,\false\}}
\newcommand{\Integer}{\mathbb{Z}}
\newcommand{\Real}{\mathbb{R}}
% --- END - HEADER TO SUPPORT TexSci ALGORITHMS IN LaTeX DOCUMENTS
```

FIGURE 2 – Code LaTeX à reporter en début de fichier .tex pour le support de TexSci

Les éléments d'une suite de déclarations sont séparés par une virgule « , ». Lorsqu'il n'y a aucune déclaration, il est possible soit de ne pas mettre de commande, soir de réduire la partie <suite de déclarations> à `\emptyset`.

Une déclaration de variable est de la forme `<nom de variable> \in <type>` où le type peut être scalaire : `<type_scalaire>` (`\Integer`, `\Real` ou `\Boolean`) ou bien vectoriel (tableau 1D) auquel cas le type est de la forme `<type_scalaire>^{<dimension>}`. Une déclaration de constante est de la forme `<nom de constante> = <valeur> \in <type_scalaire>`.

2.2.2 Zone des instructions

La zone des déclarations est suivie de la zone des instructions. Elle contient les instructions et les structures de contrôle composant l'algorithme. Par exemple on note :

- `$<nom de variable> \leftarrow <expression>$` pour une instruction d'affectation de la valeur d'une expression à une variable,
- `<instruction> \; ... \; <instruction> \;` pour une séquence de plusieurs instructions (les instructions étant terminées par `\;`),
- `\While{$<condition>$}{<instruction>}` pour une boucle while,
- `\For{$<nom de variable> \leftarrow <expression>$ \KwTo $<expression>$}{<instruction>}` pour une boucle for itérative,
- `\If{$<condition>$}{<instruction>}` pour une conditionnelle,
- `\eIf{$<condition>$}{<instruction>}{<instruction>}` pour une alternative.

Un algorithme peut faire appel à d'autres algorithmes et peut aussi être récursif. Un appel à un algorithme s'écrit `\mbox{<nom d'algorithme>($<liste d'expressions>$)}` où les expressions de la liste (éventuellement vide) sont séparées par une virgule. Cependant il vous

est demandé de vous concentrer sur le reste avant de tenter d'ajouter le support des appels à d'autres algorithmes. Dans un premier temps, seul l'algorithme `main` sera présent.

Pour la syntaxe des `<expressions>` et `<condition>` se reporter à la documentation web sur LaTeX concernant l'écriture des expressions arithmétiques ou logiques en mode mathématique (par exemple, l'opérateur de multiplication se note `\times`, les opérateurs booléens sont `\vee` (« ou » logique), `\wedge` (« et » logique), `\neg` (« non » logique), les opérateurs relationnels sont `\geq` (« plus grand ou égal »), `\leq` (« plus petit ou égal »), `\neq` (« différent »), etc. S'inspirer aussi des exemples d'algorithmes fournis ou que l'on peut trouver sur le web.

2.2.3 Bibliothèque standard

La bibliothèque standard TexSci ne fournit que trois fonctions :
— *printText()* prend une chaine de caractères ASCII en argument et l'affiche,
— *printInt()* prend un entier en argument et l'affiche,
— *printReal()* prend un réel en argument et l'affiche.
Aucune inclusion de bibliothèque n'est nécessaire pour pouvoir utiliser ces fonctions.

2.2.4 Sémantique

Dans un algorithme TexSci, les variables globales sont toujours initialisées à 0 par défaut. Il est interdit d'utiliser une variable non initialisée dans une expression.

2.3 Compiler le fichier LaTeX en un fichier PDF

Puisque LaTeX est un langage, un texte écrit en LaTeX peut être vérifié puis compilé en un document PDF présentant les algorithmes sous une forme plus lisible (comme en figure 1(b)). Les fichiers contenant un texte écrit en LaTeX ont pour extension .tex. LaTeX est gratuit et peut être installé [2] sur la plupart des plate-formes (Linux, Mac OS, Windows). La commande pour obtenir un document pdf, disons doc.pdf, à partir d'un texte source LaTeX, disons doc.tex, est :
`pdflatex doc.tex`
C'est de cette manière qu'a été compilé le présent document et les exemples disponibles sous Moodle. Pour compiler un document comme indiqué ici, vous aurez éventuellement besoin (selon votre installation de LaTeX) de placer le fichier algorithm2e.sty dans le répertoire courant (fourni sur Moodle si besoin).

3 But du projet

Lisez attentivement cette partie car votre compilateur sera testé de manière automatique : il y a des noms et des conventions à respecter absolument.

L'objectif de ce projet est d'implémenter un compilateur pour les documents contenant des algorithmes au format TexSci. Le compilateur devra être écrit en C à l'aide des outils Lex et Yacc. Le nom d'exécutable de votre compilateur devra être `texcc` (nom imposé). Si le fichier d'entrée est correct, le compilateur devra produire un code exécutable MIPS correspondant aux algorithmes d'entrée dans un fichier `output.s` (nom imposé). Si le fichier d'entrée n'est pas correct, le compilateur devra afficher un message d'erreur explicite et :
— retourner 1 (par `exit(1);`) en cas d'erreur lexicale (une partie du fichier d'entrée ne peut être ignoré ou identifié comme un token),
— retourner 2 (par `exit(2);`) en cas d'erreur de syntaxe (fichier mal formé par rapport à la grammaire),

2. Pour installer LaTeX sur votre système : `https://fr.wikibooks.org/wiki/LaTeX/Installer_LaTeX`

— retourner 3 (par `exit(3);`) en cas d'erreur sémantique (problème de type ou d'initialisation de variables),
— retourner 4 (par `exit(4);`) pour tout autre problème.

Ce travail est à réaliser en binôme et à rendre à la date indiquée par vos enseignants en cours et sur Moodle. Une démonstration de votre compilateur sera faite durant la dernière séance de TP. Vous devrez rendre sur Moodle dans une archive :

— Le code source de votre projet complet dont la compilation devra se faire simplement par la commande « `make` » pour produire l'exécutable `texcc`.
— Un court document détaillant les capacités de votre compilateur, c'est à dire ce qu'il sait faire ou non. Soyez honnêtes, indiquez bien les points intéressants que vous souhaitez que le correcteur prenne en compte car il ne pourra sans doute pas tout voir dans le code.
— Un jeu de tests.

3.1 Assembleur MIPS

Le code généré devra être en assembleur MIPS R2000. L'assembleur est décrit dans les documents `spim_long.pdf` et `spim_short.pdf` accessibles sur Moodle. Des exemples de codes MIPS sont également fournis sur Moodle.

Le code assembleur devra être exécuté à l'aide du simulateur de processeur R2000 SPIM. Celui-ci est installé sur la machine `turing`. Pour exécuter un code assembleur, il suffit de faire : `spim <nom_du_code>.s`. Si vous désirez installer SPIM sur votre propre machine :

— Téléchargez le package `spim.tar.gz` sur Moodle
— `tar xvfz spim.tar.gz`
— `cd spim-8.0/spim`
— `./Configure` (Vérifiez qu'il n'y a pas de messages d'erreur : bibliothèque ou logiciel manquant.)
— Modifiez dans le fichier `Makefile` la ligne `EXCEPTION_DIR =` en y saisissant le chemin d'accès au répertoire contenant le fichier `exceptions.s`. Il s'agit du répertoire : `/chemin_vers_spim/spim-8.0/CPU`
— `make`

3.2 Précisions concernant la notation

— Si votre projet ne compile pas ou plante directement, la note 0 (zéro) sera appliquée : l'évaluateur n'a absolument pas vocation à aller chercher ce qui pourrait éventuellement ressembler à quelque chose de correct dans votre code.
— Si vous manquez de temps, préférez faire moins de chose mais en le faisant bien et de bout en bout : on préfèrera un support de StenC incomplet mais qui génère un code assembleur exécutable à une analyse syntaxique seule.
— Élaborez des tests car cela fait partie de votre travail et ce sera donc évalué.
— Faites les choses dans l'ordre et focalisez sur ce qui est demandé. L'évaluateur pourra tenir compte du travail fait en plus (optimisations du code généré par exemple) seulement si ce qui a été demandé a été fait et bien fait.
— Une conception modulaire et lisible sera fortement appréciée (et inversement). Une bonne démarche est de séparer les différentes étapes de la compilation comme cela a été vu en cours, en définissant des formats intermédiaires (par exemple on préfèrera une génération de code sous forme de quads génériques, suivie de la traduction des quads en assembleur, plutôt qu'une génération assembleur directe). Étant donnée l'ampleur de la tâche, on ne demande pas le passage par un arbre de syntaxe abstraite entre analyse syntaxique et génération de code, mais ceux qui le feront correctement seront des héros.

3.3 Recommandations

Conseil important : écrire un compilateur est un projet conséquent, il doit donc impérativement être construit **incrémentalement** en validant chaque étape sur un plus petit langage et en ajoutant progressivement des fonctionnalités ou optimisations (à ce propos, les optimisations du compilateur sont secondaires : si vous écrivez du code modulaire, il sera toujours possible par exemple de changer la représentation de la table des symboles en utilisant une structure de données plus efficace).

Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis passer au débogage ! Le résultat de cette démarche serait très probablement nul, c'est-à-dire un compilateur qui ne fonctionne pas du tout ou alors qui reste très bogué.