Université de Strasbourg

Master Informatique 1ère année
Compilation - Projet 2020

Compilateur pour le langage **SCALPA**

But du projet : Réaliser un compilateur pour le langage SCALPA décrit ci-dessous. Ce compilateur produira en sortie du code assembleur MIPS qui devra s'exécuter sans erreurs à l'aide d'un simulateur de microprocesseur MIPS.

1 Présentation du langage

1.1 Grammaire du langage

On donne ci-dessous la grammaire complète du langage SCALPA. Dans cette grammaire, les mots-clés et les terminaux apparaissent en fonte courrier. Le terminal `cte` représente les constantes entières, booléennes (`true` et `false`), ou chaînes de caractères. Le terminal `ident` représente les identificateurs qui sont formés d'un caractère alphabétique suivi d'une suite de caractères alphanumériques, ou d'une apostrophe, ou du caractère de soulignement. Les commentaires débutent par les deux caractères `(*` et se terminent par les deux caractères `*)`. Ils peuvent être imbriqués. Le type `unit` sert à spécifier le type de la valeur retournée par une fonction dont le seul but est de causer un effet de bord, ainsi que le type d'argument d'une fonction qui ne requiert pas d'argument (comme le type `void` en C).

program	→	`program ident` vardecllist fundecllist instr
vardecllist	→	ϵ | varsdecl | varsdecl `;` vardecllist
varsdecl	→	`var` identlist `:` typename
identlist	→	`ident` | `ident ,` identlist
typename	→	atomictype | arraytype
atomictype	→	`unit` | `bool` | `int`
arraytype	→	`array [` rangelist `] of` atomictype
rangelist	→	integer `..` integer | integer `..` integer `,` rangelist
fundecllist	→	ϵ | fundecl `;` fundecllist
fundecl	→	`function ident (` parlist `) :` atomictype vardecllist instr
parlist	→	ϵ | par | par `,` parlist
par	→	`ident :` typename | `ref ident :` typename
instr	→	`if` expr `then` instr | `if` expr `then` instr `else` instr
	|	`while` expr `do` instr | lvalue `:=` expr | `return` expr | `return`
	|	`ident (` exprlist `)` | `ident ( )` | `begin` sequence `end` | `begin end`
	|	`read` lvalue | `write` expr
sequence	→	instr `;` sequence | instr `;` | instr
lvalue	→	`ident` | `ident [` exprlist `]`
exprlist	→	expr | expr `,` exprlist
expr	→	`cte` | `(` expr `)` | expr opb expr | opu expr
	|	`ident (` exprlist `)` | `ident ( )` | `ident [` exprlist `]` | `ident`
opb	→	+ | − | ∗ | / |ˆ| < | <= | > | >= | = | <> | `and` | `or` | `xor`
opu	→	− | `not`

Quelques précisions :

- Cette grammaire pourra être légèrement modifiée en une autre grammaire équivalente au besoin.
- La priorité suivante des opérateurs devra être respectée (ordre décroissant de priorité) :
  1. `-` (unaire), `not`
  2. `^`

3. *, /, `and`
4. +, `-`, `or`, `xor`

— Les intervalles d'indices de tableau sont inclusifs. Par exemple, l'intervalle [-4..2] signifie que les indices du tableau sont -4, -3, -2, -1, 0, 1 et 2.
— Une déclaration de paramètre de fonction avec le mot clé `ref` permet d'effectuer un passage de paramètre *par référence* à la fonction. Ainsi, toute modification de la valeur du paramètre par la fonction sera visible par l'appelant.
— Les tableaux peuvent être passés par valeur aux fonctions.
— Les constantes chaînes de caractères sont délimitées par des double-quotes ", comme dans `"Ceci est une chaîne"`.
— Les constantes booléennes sont `true` et `false`, reconnues lors de l'analyse lexicale.
— Les opérateurs $+$, $-$, $*$, $/$, $\hat{}$, $<$, $<=$, $>$ et $>=$ ne sont utilisables qu'avec des opérandes qui sont des expressions arithmétiques.
— Les opérateurs logiques `and`, `or`, `xor` et `not` ne sont utilisables qu'avec des opérandes qui sont des expressions booléennes.
— Bien qu'il y ait des constantes chaînes de caractères, il n'y a pas d'opérations possibles sur ces constantes.

1.2 Exemple de programme

```
(* Ceci est un exemple de code SCALPA *)
program maxi
  var i,j,max : int;
  var val : array[-4..4, 0..10] of int

  function fini() : unit
  begin
    write "programme terminé"
  end;
  function maxtab(t : array[-4..4, 0..10] of int) : int
    var i,j,maximum : int
    begin
      i := -4;
      j := 0;
      maximum := t[-4,0];
      while i <= 4 do
      begin
        while j <= 10 do
        begin
          if t[i,j] > maximum then maximum := t[i,j];
          j := j+1
        end;
        i := i+1;
      end;
      return maximum
    end;
begin
  i := -4;
  j := 0;
  while i <= 4 do
  begin
    while j <= 10 do
    begin
      read val[i,j];
      j := j+1
    end;
    i := i+1
  end;
  max := maxtab(val);
  write max;
  fini()
end
```

2 Génération de code

Le code généré devra être en assembleur MIPS R2000. L'assembleur est décrit dans les documents fournis. Le code assembleur devra être exécuté à l'aide du simulateur de processeur MIPS *SPIM*[1] (il existe un package debian/ubuntu) ou *Mars*[2].

3 Aspects pratiques et techniques

Le compilateur devra être écrit en C à l'aide des outils Lex et Yacc.

Ce travail est à réaliser en équipe composée de quatre étudiant.e.s dans le cadre du cours de *Compilation* et du cours de *Conduite de Projets*, et à rendre à la date indiquée par vos enseignants en cours et sur Moodle. Une démonstration finale 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 ». Le nom de l'exécutable produit doit être « scalpa »
- Un 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.

Votre compilateur devra fournir les options suivantes :

- `-version` devra indiquer les membres du projet.
- `-tos` devra afficher la table des symboles.
- `-o <name>` devra écrire le code résultat dans le fichier `name`.

4 Recommandations importantes

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. Une démarche extrême et totalement contre-productive consiste à écrire la totalité du code du compilateur en une fois, puis de 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.

Par conséquent, nous vous conseillons de développer tout d'abord un compilateur *fonctionnel* mais *limité* à la traduction d'expressions arithmétiques simples, sans structures de contrôle. À partir d'une telle version fonctionnelle, il vous sera plus aisé de la faire évoluer en intégrant telle ou telle fonctionnalité, ou en considérant des expressions plus complexes, ou en intégrant telle ou telle structure de contrôle. De plus, même si votre compilateur ne remplira finalement pas tous les objectifs, il sera néanmoins capable de générer des programmes corrects et qui «marchent» !

5 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. Il faut que votre compilateur s'exécute et qu'il fasse quelque chose de correct, même si c'est peu.
- Si vous manquez de temps, préférez faire moins de choses mais en le faisant bien et de bout en bout : on préférera un compilateur incomplet mais qui génère un programme MIPS exécutable plutôt qu'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 (par exemple des optimisations de code) seulement si ce qui a été demandé a été fait et bien fait.
- Une conception modulaire et lisible sera fortement appréciée (et inversement).

1. `http://spimsimulator.sourceforge.net`
2. `http://courses.missouristate.edu/kenvollmar/mars`