Génération de code pour `tinyC` avec ANTLR4

ANTLR

ANTLR4 est un générateur d'analyseur récursif LL (analyse descendante) qui est capable de supprimer la récursivité gauche immédiate, et permet aussi comme CUP de spécifier des règles d'associativité pour supprimer les ambiguités de priorité. L'écriture des règles de grammaire reste ainsi assez simple (ce qui est un des avantages de l'analyse ascendante). Il est écrit en Java et permet de générer des analyseurs dans différents autres languages (Javascript et Python notamment).

Nous utilisons une grammaire préexistante pour un sous ensemble du langage C, `tinyC`.

```
<program> ::= <statement>
<statement> ::= "if" <paren_expr> <statement> |
                "if" <paren_expr> <statement> "else" <statement> |
                "while" <paren_expr> <statement> |
                "do" <statement> "while" <paren_expr> ";" |
                "{" { <statement> } "}" |
                <expr> ";" |
                ";"
<paren_expr> ::= "(" <expr> ")"
<expr> ::= <test> | <id> "=" <expr>
<test> ::= <sum> | <sum> "<" <sum>
<sum> ::= <term> | <sum> "+" <term> | <sum> "-" <term>
<term> ::= <id> | <int> | <paren_expr>
<id> ::= "a" | "b" | "c" | "d" | ... | "z"
<int> ::= <an_unsigned_decimal_integer>
```

Le fichier `tinyC.g4` du dépôt contient la grammaire de ce langage au format `ANTLR4`. Nous créons l'analyseur correspondant ainsi:

```
mkdir javabuild
antlr4 tinyc.g4 -visitor -o javabuild
cd javabuild
javac -cp /usr/share/java/antlr4-runtime.jar:. *.java
```

Cela construit et compile l'ensemble des classes pour obtenir un analyseur pour la grammaire de `tinyC`. Vous pouvez les tester ainsi (copier-coller le petit morceau de C des exemples, suivi de `<Ctrl-D>`): * Reconnaissance des tokens (option `-tokens`)

```
/usr/share/antlr4/grun tinyc  program -tokens
{ i=7; if (i<5) x=1; if (i<10) y=2; }
```

- Affichage de l'arbre de syntaxe abstraite (option `-gui`)

  ```
  java -cp .:/usr/share/java/antlr4.jar:/usr/share/java/antlr4-runtime.jar:/usr/share/j
     org.antlr.v4.gui.TestRig tinyc program -gui
  { i=7; if (i<5) x=1; if (i<10) y=2; }
  ```

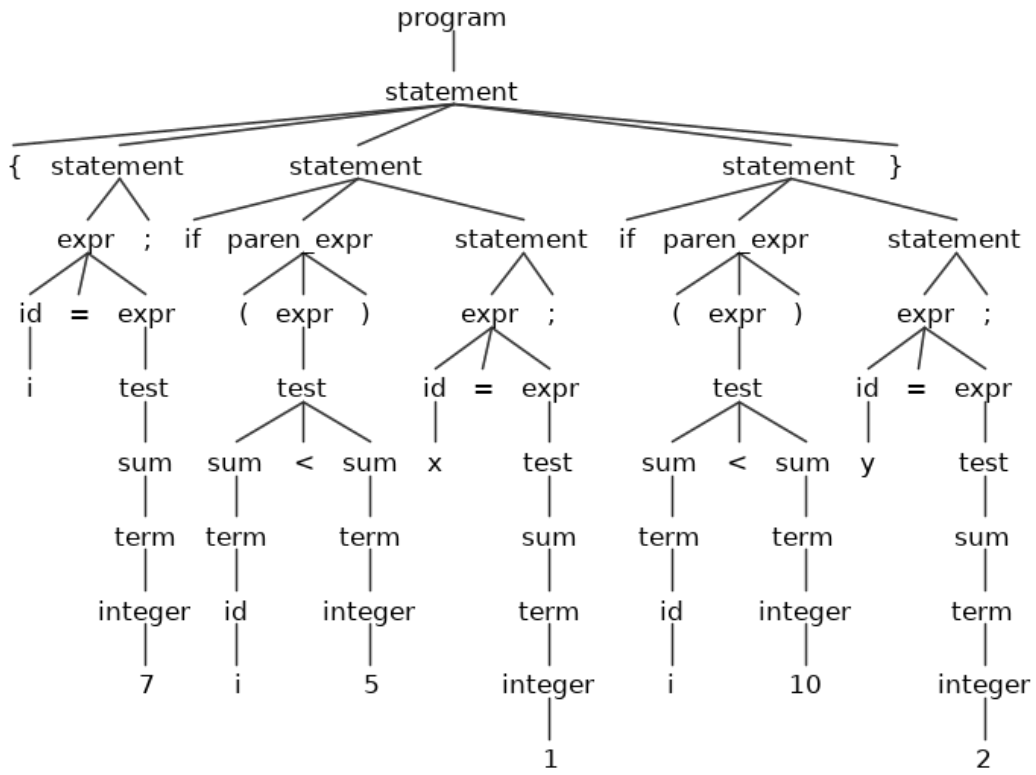

Figure 1: AST pour le programme `{ i=7; if (i<5) x=1; if (i<10) y=2; }`

Python 3

Nous allons utiliser un wrapper pour pouvoir développer en Python 3. Pour réaliser le même test que ci-dessus nous utilisons le programme `pgrun` disponible sur le site de développement du wrapper et fourni dans ce dépôt. Notez que la grammaire a été modifée car certaines variables de la grammaire initiale (`id` et `sum`) sont des mots clé de Python et `antlr4` va générer des procédures utilisant ces noms de variable (analyse récursive).

```
pip3 install --user antlr4-python3-runtime
mkdir pythonbuild
antlr4 pytinyc.g4 -Dlanguage=Python3 -visitor -o pythonbuild
cd pythonbuild
python3 ../pygrun -k pytinyc program   # tokens
{ i=7; if (i<5) x=1; if (i<10) y=2; }
python3 ../pygrun -t pytinyc program   # pretty-print (pas d'option gui)
{ i=7; if (i<5) x=1; if (i<10) y=2; }
```

Assembleur MIPS

Le but est de générer du code pour un processeur d'architecture MIPS1, une architecture RISC 32 bits développée par MIPS Technologies. La particularité des processeurs RISC est que toutes les opérations doivent être réalisées entre des registres, aucune opérande ne peut se situer en mémoire. Cela nécessite donc de réaliser explicitement les transferts entre mémoire et registres (on parle d'architecture Load/Store). Le processeur dispose de 32 registres de 32 bits, leur usage est présenté dans la table ci-dessous. Notez que `$zero` est en lecture seule, et qu'il faut respecter cet usage des registres pour que le simulateur fonctionne. Nous n'utiliserons que les registres temporaires `$t0` et `$t1`, ainsi que le pointeur de pile `$sp`.

Number	Name	Use	Preserved across function calls?
0	`$zero`	constant 0	—
1	`$at`	assembler temporary	no
2, 3	`$v0, $v1`	function return values	no
4 - 7	`$a0 - $a3`	function arguments	no
8 - 15	`$t0 - $t7`	temporaries	no
16 - 23	`$s0 - $s7`	temporaries	yes
24, 25	`$t8, $t9`	temporaries	no
26, 27	`$k0, $k1`	reserved for OS kernel	—
28	`$gp`	global pointer	—
29	`$sp`	stack pointer	—
30	`$s8`	temporaries	yes
31	`$ra`	return address	—

Le jeu d'instructions du processeur MIPS est résumé dans le document MIPS32 Instruction Set Quick Reference et spécifié dans sa totalité dans The MIPS32 Instruction Set(485 pages!). Les instructions nécessaires pour le moment sont les suivantes (`RD`, `RS` et `RT` désignent chacun

l'un des registres ci-dessus):

Arithmetic Operations	Sémantique	Utilisation
`ADD RD, RS, RT`	RD = RS + RT (OVERFLOW TRAP)	opérateur `+`
`ADDIU RD, RS, CONST16`	RD = RS + CONST16 ±	incrément/décrément de `$sp`
`LI RD, IMM32`	RD = IMM32	constante `integer`
`SUB RD, RS, RT`	RD = RS – RT (OVERFLOW TRAP)	opérateur `-`
Logical and Bit-Field Operations		
`XOR RD, RS, RT`	RD = RS ⊕ RT	opérateur `==`
Conditional Move Operations		
`SLT RD, RS, RT`	RD = (RS± < RT±) ? 1 : 0	opérateurs `<` et `>`
Jumps and Branches	**(Note: One Delay Slot)**	
`BEQZ RS, OFF18`	IF RS = 0, PC += OFF18 ±	structures `if`/`if-else`
`B  OFF18`	PC += OFF18 ±	structures `if-else`
`NOP`	No operation	delay Slot
Load and Store Operations		
`LW RD, OFF16(RS)`	RD = MEM32(RS + OFF16 ±)	variable `varid`
`SW RS, OFF16(RT)`	MEM32(RT + OFF16±) = RS	Affectation `varid =`

Nous utilisons le simulateur MARS développée par l'Université du Missouri (pour l'enseignement entre autres). Il contient un assembleur MIPS et son environnement complet d'exécution (visualisation du programme, des registres et de la mémoire, et exécution pas-à-pas). Il est distribué sous forme d'achive `jar`, et est inclus dans ce dépôt. Vous lancez son exécution par la commande `java -jar Mars4_5.jar`. Essayez-le en chargeant le programme `min-max.asm`, puis en assemblant ce programme source, et enfin en lancant l'exécution du binaire obtenu. La fenêtre d'aide du simulateur vous rappelle l'ensemble des instructions implémentées (dont les instructions flottantes) et les appels systèmes disponibles (kernel).

Génération de code

Le fichier `tinyCc.py` contient le squelette d'un compilateur pour `tinyC` basé sur `ANTLR4`. Ce programme prend simplement un nom de fichier source en paramètre, il s'exécute par la commande `python3 tinyCc.py fichier.c` depuis le répertoire de travail (au dessus de `pythonbuild`). Il produit son résultat, un fichier source assembleur MIPS, sur la sortie standard. Vous pouvez la rediriger vers `fichier.asm` que vous chargerez dans le simulateur `MARS` où il ne restera plus qu'à l'assembler et à tester son exécution. Vous pouvez essayer de compiler et exécuter le fichier `examples5.c`.

Le compilateur tel qu'il vous est fourni implémente les phases d'analyse lexicale (`class pytinycLexer`) et d'analyse syntaxique (`class pytinycParser`) dans la fonction `compile`. Voir le fichier `pygrun` et la documentation de l'API d'`ANTLR4` pour leurs fonctionnements détaillés. L'AST obtenu (`parse_result`) est alors passé en paramètre à la fonction `gencode` (en plus du flux de tokens pour accéder à leur valeur littérale). Cette fonction réalise un parcours récursif de cet arbre et renvoie le code assembleur généré sous forme d'une chaîne de caractères. La structure de la fonction `gencode` est un aiguillage `(if/elif/elif...)`

dépendant de la règle associée au noeud courant de l'arbre (accessible dans le contexte de la règle de type `ParserRuleContext`, paramètre `tree`) .

- Règle `program`: c'est la racine de l'arbre. Le code généré est composé de la concaténation du code généré par tous les fils du noeud, auquel on ajoute quelques directives assembleur constantes. Il faut aussi ajouter après parcours des fils, les déclarations des variables dans l'assembleur. Cette déclaration se fait dans le `.data` segment en parcourant la table des symboles (dictionnaire `symbols`) remplie pendant le parcours des fils.
- Règle `statement`: l'examen du premier fils de cette règle (`tree.getChild(0)`) permet de déterminer le type de l'instruction. On réalise à nouveau un aiguillage:
- on génère la chaîne vide pour un `statement` vide (`;`)
- on concatène le code généré par tous les fils du noeud pour un bloc `{ statement }`
- pour les structures `if` et `if/else`, on génère le code pour évaluer l'expression conditionnelle (`tree.getChild(1)`). Le résultat de cette évaluation est, par construction de le fonction `gencode`, placé dans le registre `$t0$`. Si `$t0` vaut zéro, le code généré doit effectuer un saut vers une étiquette placée juste après le code généré par le bloc du `if`: cette étiquette est construite par concaténation de la chaîne `label_endif_` à laquelle on ajoute le numéro de ligne et le numéro de colonne où est située le `if` dans le code source (les étiquettes doivent être uniques). Pour un `if` on génère alors l'instruction `BEQZ $t0, label_endif_line_col` puis le code généré par le bloc du `if` (`tree.getChild(2)`), et on place l'étiquette `label_endif_line_col` à la fin du bloc. Pour un `if/else` le principe est similaire (étudiez le code sur un exemple).
- Règle `expr`:
- si il s'agit d'une expression d'assignation (3 fils), on génère le code pour évaluer le membre droit de l'assignation (`tree.getChild(2)`), le résultat qui se trouve dans `$t0` doit alors être transféré en mémoire par une instruction `SW $t0, varid` (`varid` étant la valeur textuelle du premier fils du noeud). On consulte la table des symboles pour éventuellement y adjoindre `varid` si la variable en est absente.

- sinon il s'agit d'évaluer une expression arithmétique et logique de profondeur et de largeur quelconques. Disposant d'un nombre de registres limité, la problématique de générer un code optimal est complexe. Nous choisissons une méthode naive qui consiste à utiliser un registre (`$t0`) pour évaluer le premier membre de l'expression, d'empiler alors sa valeur (`push $t0`), de réutiliser ce même registre pour évaluer le second membre de l'expression, de dépiler dans un second registre (`pop $t1`) la valeur du premier membre pour finalement effectuer l'opération entre `$t0` et `t$1`. Noter que les architectures RISC n'ont pas toujours d'instructions (`push/pop`), ces opérations se réalisent avec des `load/store` (adressage indirect par registre avec déplacemnent `disp($sp)`) suivie/précédée d'une incrémentation/décrémentation du pointeur de pile `$sp`. C'est le cas du MIPS, avec par convention une pile croissante vers les adresses basses de la mémoire.

Question 1

Ajouter la génération de code pour les instructions `while` et `do/while`. Faites un test avec le programme suivant (`fibonacci.c`).

```
n=12; u0 = 1; u1 = 1; i = 1;
while (i < n) {temp = u1; u1 = u1 + u0; u0 = temp; i = i + 1;}
```

Question 2

Ajouter à la grammaire l'appel système `printinteger(expr)` en modifiant la grammaire ainsi:

```
statement
   ...
   | SEMICOLON
   | PRINTINTEGER LPAR expr RPAR
   ;
   ...
   PRINTINTEGER
      :
      'printinteger'
      ;
```

La génération de code pour un appel système se fait en spécifiant le numéro d'interruption dans le registre `$v0` (`li $v0, 1`), l'argument de l'appel dans le registre `$a0` (`move $a0, $t0`) et en exécutant l'instruction d'interruption logicielle (`syscall`). Voir l'aide de `MARS`. Tester en ajoutant l'affichage du résultat dans le programme `fibonacci.c`.

Pour aller plus loin (Facultatif)

Vous pouvez ajouter de nouvelles règles à la grammaire de `tinyC` en vous basant sur une grammaire du langage C telle celle proposée ici. La génération de code pour ces ajouts peut cependant devenir rapidement complexe.

Question 3

Ajouter la déclaration et le typage des variables en modifiant la grammaire ainsi:

```
program: declaration* statement +;
statement
    ....
```

```
     LCBRACKET declaration* statement* RCBRACKET
     // à la place de LCBRACKET statement* RCBRACKET
    ....
declaration : typespecifier varid SEMICOLON;
typespecifier : INTTYPE | DOUBLETYPE ;
term: ... | integer | double
double : DOUBLE | DOUBLEHEX ;
INTTYPE: 'int';
DOUBLETYPE: 'double';
DOUBLE
   :
     ([0-9]*.[0-9]+|[0-9]+'.')([eE][+-]?[0-9]+)?[flFL]?
   | [0-9]+[eE][+-]?[0-9]+[flFL]?
   ;
DOUBLEHEX
   :
     '0'[xX]([0-9a-fA-F]*'.'[0-9a-fA-F]+ | [0-9a-fA-F]+'.')[pP][+-]?[0-9]+[flFL]?
   | '0'[xX][0-9a-fA-F]+[pP][+-]?[0-9]+[flFL]?
   ;
```

Au moment de la génération de code il faudra vérifier qu'une variable est effectivement déclarée, et éventuellement prendre en compte le transtypage par défaut du C. Notez qu'il faut utliser une table de symboles globale et des tables de symboles locales aux différents blocs d'instruction. Les variables locales sont allouées dans la pile, leurs adresses (déplacement par rapport à `$sp`) sont mémorisées dans la table des symboles pendant la génération de code. Pour le transtypage, le processeur MIPS dispose d'instructions de conversion (`cvt`, voir l'aide de `MARS`). Dans un premier temps vous pouvez ne considérer que la déclaration de variables (entières).

Question 4

Ajouter la définition et l'appel de fonctions en modifiant la grammaire ainsi par exemple

```
functiondefinition: typespecifier varid LPAR parameterlist RPAR
                        LCBRACKET declaration* statement* RCBRACKET;
parameterlist : | typespecifier varid parametertail;
parametertail: COLON typespecifier varid parametertail | ;
...
expr:
   ...
   | varid LPAR expr RPAR
   ...
COLON : ',';
```

La définition d'une fonction se fait en ajoutant une étiquette dans le code généré, marquant le point d'entrée de la fonction. A la fin de la fonction on revient à l'appelant en générant le code permettant de restaurer le compteur de programme `PC`. Il faut utiliser la pile pour sauvegarder ces adresses de retour. De même les valeurs des paramètres effectifs et la valeur de retour sont généralement aussi passés par la pile. Cependant sur les processeurs RISC certains registres sont réservés à cet usage (moins de 4 paramètres, une seule valeur de retour) ce qui évite l'utilisation de la pile. On peut utiliser ce mode de passage de paramètres pour les fonctions terminales qui n'appellent pas d'autres fonctions par exemple.

---

1 On peut trouver des processeurs MIPS dans des routeurs, des NAS ou des consoles (PS, PS2, PSP) par exemple. Cette architecture bénéficie d'un regain actuellement, la Chine et la Russie l'utilisant dans leurs projets de supercalculateur.