Quiz

Compiler Construction

December 17, 2008

Last Name : _____

First Name : _____

Exercise	**Points**	**Achieved Points**
1	20	
2	20	
3	20	
4	20	
Total	80	

General advice (it may or may not work for you):

— Do not panic.
— Read all assignments first.
— Note the total amount of time you have.
— First solve those problems that you think you know well or that you expect to be easy.
— If you do not know how to solve a part of a problem, you can leave it for later.

The description of ϵ-lang

The questions of the quiz revolve around a small functional language for arithmetic expressions which we call ϵ-lang and of which we give an informal overview here. The grammar of the language is as follows:

```
program   ::= ex
ex        ::= ex + ex | ex * ex | ex / ex | block | ( ex ) | ex ( ex (, ex)* )
            | ex || ex | ex < ex | ident | intLiteral | floatLiteral
block     ::= { defn* ex }

defn      ::= valDef | fDef
valDef    ::= val ident : type = ex
fDef      ::= def ident ( typedIdList ) : type = ex
typedIdList ::= ident : type (, ident : type)*
type      ::= int | float | ( type (, type)* ) ⇒ type | ( type )
```

The tokens of the language include keywords and symbols, as well as the token classes given by the following grammar:

```
ident       ::= letter (letter | digit)*
intLiteral  ::= decIntLit | hexIntLit | octIntLit
decIntLit   ::= nonZeroDigit digit* | 0
hexIntLit   ::= 0x hexDigit+
octIntLit   ::= 0 digit+
floatLiteral ::= decIntLit . digit*
letter      ::= [a-zA-Z]
digit       ::= [0-9]
nonZeroDigit ::= [1-9]
hexDigit    ::= [0-9a-f]
```

— ϵ-lang has two primitive types, `int` and `float`, and functional types. For instance, the type `(int, int)` ⇒ `float` represents functions which take two integers and compute a float. All functions take at least one argument.
— Integers can be input in either decimal, hexadecimal or octal format, and the input format doesn't affect the type of the literal —they are all `int`. For example, `42 = 0x2a = 052`. Note that the language does not allow the expression of negative values.
— The operator priorities from most tightly binding to least tightly binding are: `*` and `/`, `+`, `<`, || (`*` and `/` have the same priority).
Operators `*`, `/`, `+`, || are left-associative, whereas `<` is non-associative.
For example:

```
x < y + z * u * v || p
```

... corresponds to the fully parenthesized expression:

```
(x < (y + ((z * u) * v))) || p
```

— The operator || takes as arguments two integer expressions. If the first argument is zero, the result is the second argument. If the first argument is not a zero, then the result is the first argument and the second one is not evaluated.

— The operators `+`, `*` and `/` are overloaded for integers and floats with the following rules:
  — If both operands are of type `int`, the result is of type `int`.
  — If both operands are of type `float`, the result is of type `float`.
  — If one operand is of type `int` and the other is of type `float`, the integer value is converted to the corresponding floating-point value and the result is a `float`.

  Division between two integers is the usual truncating operator; if, on the other hand, one of the operators is a `float`, the result is given by floating point division.

— The operator `<` takes as arguments two integer expressions and returns 1 if the left-hand side value is smaller than the right-hand side one, and 0 otherwise. The result is of type `int`.

— Programs in ϵ-lang consist of a single expression, but expressions can be blocks and blocks can contain definitions of functions and constants. A block evaluates to the expression it contains.

— There cannot be two functions with the same name defined in the same block (functions defined with **`def`** cannot be overloaded).

As an example, consider the following ϵ-lang program which computes the factorial of 10:

```
{
    val goal : int = 10
    def fact(x : int) : int = {
        def inner(i : int) : int = {
            x < i || i * inner(i + 1)
        }
        inner(1)
    }
    fact(goal)
}
```

Exercise 1 : Lexical Analysis (20 points)

Describe a lexical analyzer that recognizes a sequence of the following token classes of ϵ-lang:

- identifiers
- decimal integer literals
- hexadecimal integer literals
- octal integer literals
- floating point literals

. . . and skips whitespace characters between the tokens. (Denote the whitespace characters by the $\sqcup$ symbol.)

Present your lexical analyzer as a finite state machine that has different accepting states for the different token classes. Make sure to implement the longest-match rule.

You can use ranges of characters, such as [a-z], to label your transitions, but do not use the regular expression operators `|` , `*`, or `+` (you need to express those operators using appropriate states and transitions in the automaton).

Exercise 2 : Parsing (20 points)

1. Consider the ϵ-lang grammar. Compute:
   (a) the set of nullable non-terminals
   (b) the set $first(\textit{defn})$
   (c) the set $follow(\textit{block})$
2. Give at least one specific reason why the given grammar does not belong to the class $LL(1)$ (list the relevant values of $nullable$, $first$, $follow$ and the relevant grammar rules).
3. Rewrite the rule for expressions *ex* into several equivalent grammar rules that encode the priorities and associativity of operators `*`, `/`, `+`, `<`, ||, and eliminate left-recursion from the definition of *ex*.
4. Using your new grammar, draw the concrete syntax tree for the expression:

```
a + y < p / z / w * u
```

5. Note that expressions in our language can use parentheses freely, and that, for a function f of type:

```
(int) ⇒ ((int) ⇒ int)
```

   ...our language allows us to write expressions such as `f(x)(y)`, which is equivalent to:

```
{
    val resf : (int) ⇒ int = f(x)
    val res : int = resf(y)
    res
}
```

   Explain what problems such syntax presents for an $LL(1)$ parser. Suggest a small change to the concrete syntax which would solve this.

Exercise 3 : Type Checking (20 points)

1. Provide type checking rules for the following constructs of our language:
   (a) division operator (`/`)
   (b) less-than operator (`<`)
   (c) function application
   (d) function definitions (*fDef*)
   (e) block ({ ... })

   Write all rules with respect to the environment Γ. You can extend Γ with the operator $\uplus$ and assume that it will correctly override the nested definitions according to the scoping rules.
   For example, you could define the rule for typechecking value definitions as follows:

   $$\textsc{Value Definition}\ \frac{\Gamma \uplus (x \mapsto T_x) \vdash \{\ ds\ ex\ \} : T}{\Gamma \vdash \{\ \texttt{val}\ x : T_x\ ds\ ex\ \} : T}$$

   ... and subsequently assume that x is available in Γ and well-typed in the rest of the block where it is defined.

   To check if a variable is present in the environment, you can write rules such as the following:

   $$\textsc{Constant lookup}\ \frac{(x \mapsto T) \in \Gamma}{\Gamma \vdash x : T}$$

   Note: You can assume that functions defined in a block are not recursive, and that you can only call a function if it is defined previously or if it is an argument.

2. Consider the following ϵ-lang program:

```
{
    def twice(f : float ⇒ float) : float ⇒ float = {
        def g(x : float) : float = {
            f(f(x))
        }
        g
    }
    def h(x : float) = 0.5 * x
    twice(h)(8 + 2.3)
}
```

   Apply your type checking rules to the expression `twice(h)(8 + 2.3)` in the context of the above program, assuming that `twice` and `h` are type correct and present in the environment. Show the entire derivation leading to the final type.

3. Modify the rule for type checking *block* to admit recursive function definitions.

Exercise 4 : Code Generation (20 points)

Consider the following code:

```
val z : int = ((x < 1) || (1000 < y / x)) + 5
```

1. Show a sequence of Java Virtual Machine code that you would generate for this ϵ-lang code.

   Assume that `x` and `y` are available as local variables of type `int` in positions 1 and 2 respectively, and that you want to store `z` at local position 3.

   If you wish, you can use the Cafebabe abstract bytecodes (as in your project).

   Use symbolic labels to denote the targets of jumps.

   Here are some bytecodes that may be useful (we do not claim that all of them are useful and we do not claim that they are sufficient):

   ```
   ILoad(slot : Int)
   IStore(slot : Int)
   IADD, IAND, IOR, IDIV, IMUL
   If_ICmpLt(target: String)
   If_ICmpGe(target: String)
   ```

2. Consider the value definition:

   ```
   val a : int = 5 + b + (c + 1) / d / 9
   ```

   ... where `b`, `c`, `d` and `e` are all of type `int`. Rewrite this value definition into an equivalent block of the following form:

   ```
   val a : int = {
     val tmp1 : int = e1
     val tmp2 : int = e2
     val tmp3 : int = e3
     val tmp4 : int = e4
     val tmp5 : int = e5
     tmp5
   }
   ```

   Each expression `e1`, `e2` ,`e3`, `e4`, `e5` should be the form $p \,???\, q$ where ??? denotes some binary operators and p, q denote constants and variables.

3. At each point in the block indicate the subset of temporary variables { `tmp1`, `tmp2`, `tmp3`, `tmp4`, `tmp5` } that are live at that point. Assume that all variables `a`, `b`, `c`, `d` are live after the block, and that no temporary variable is live after the block.

4. Draw the graph whose five nodes are temporary variables and where an edge indicates that two variables cannot be stored in the same register. What is the minimal number of colors that can be used to color the graph such that no two adjacent nodes have the same color?

END OF QUESTIONS

VERSION FRANÇAISE

La traduction française du test est donnée à titre indicatif uniquement. En cas de doute, la version anglaise fait foi.

Conseils généraux (qui peuvent ou non vous aider):

— Ne paniquez pas.
— Lisez toutes les questions avant de commencer.
— Prenez note du temps à votre disposition.
— Commencez par résoudre les problèmes qui vous paraissent facile ou qui portent sur les sujets que vous connaissez le mieux.
— Si vous n'arrivez pas à résoudre un problème en entier, laissez la fin pour plus tard.

Description de ϵ-lang

Les questions du test portent sur un petit langage fonctionnel pour des expressions arithmétiques que nous appelons ϵ-lang et dont nous donnons ici un apperçu informel. La grammaire du langage est:

```
program    ::= ex
ex         ::= ex + ex | ex * ex | ex / ex | block | ( ex ) | ex ( ex (, ex)* )
             | ex || ex | ex < ex | ident | intLiteral | floatLiteral
block      ::= { defn* ex }

defn       ::= valDef | fDef
valDef     ::= val ident : type = ex
fDef       ::= def ident ( typedIdList ) : type = ex
typedIdList ::= ident : type (, ident : type)*
type       ::= int | float | ( type (, type)* ) ⇒ type | ( type )
```

Les lexèmes (*tokens*) du langage sont les mots-clés et les symboles de ponctuation, ainsi que les classes de lexèmes représentées par la grammaire suivante:

```
ident      ::= letter (letter | digit)*
intLiteral ::= decIntLit | hexIntLit | octIntLit
decIntLit  ::= nonZeroDigit digit* | 0
hexIntLit  ::= 0x hexDigit+
octIntLit  ::= 0 digit+
floatLiteral ::= decIntLit . digit*
letter     ::= [a-zA-Z]
digit      ::= [0-9]
nonZeroDigit ::= [1-9]
hexDigit   ::= [0-9a-f]
```

— ϵ-lang comprend deux types primitifs, les entiers (`int`) et les nombres à virgule (`float`), ainsi que des types de fonctions. Par exemple, le type `(int, int)` ⇒ `float` représente les fonctions qui prennent deux entiers et retournent un nombre à virgule. Toutes les fonctions prennent au moins un argument.

— Les nombres entiers peuvent être saisis en décimal, hexadécimal ou en octal, et le format de saisie n'influence pas le type de la constante —ce sont toutes des `int`s. Par exemple, `42 = 0x2a = 052`. Notez que le langage ne permet pas d'exprimer les nombres négatifs.
— La priorité des opérateurs, par ordre décroissant, est la suivante: `*` et `/`, `+`, `<`, || (`*` et `/` ont la même priorité).
Les opérateurs `*`, `/`, `+`, || sont associatifs à gauche, et `<` est non-associatif.
Par exemple:

```
x < y + z * u * v || p
```

... correspond à l'expression suivante avec des parenthèses:

```
(x < (y + ((z * u) * v))) || p
```

— L'opérateur || prend comme arguments deux expressions entières. Si le premier argument est évalué à 0, le résultat est le deuxième argument. Si le premier argument n'est pas évalué à 0, le résultat est ce premier argument et le second n'est pas évalué.
— Les opérateurs `+`, `*` et `/` sont surchargés pour les entiers et les nombres à virgules selon les règles suivantes:
  — Si les deux opérandes sont de type `int`, le résultat est de type `int`.
  — Si les deux opérandes sont de type `float`, le résultat est de type `float`.
  — Si un des opérandes est de type `int` et l'autre de type `float`, celui qui est entier est converti en nombre à virgule et le résultat est un `float`.

  La division d'entiers suit les règles habituelles: seul le quotient de la division est gardé. Par contre, si un des opérandes est un `float`, le résultat est celui d'une division réelle.
— L'opérateur `<` prend deux expressions entières comme arguments et retourne 1 si la valeur de celle de gauche est inférieure à celle de droite et 0 sinon. Le résultat est de type `int`.
— Un program en ϵ-lang consiste en une expression unique, mais les expressions peuvent être des blocs et les blocs peuvent contenir des définitions de fonctions et de constantes. L'évaluation d'un bloc se fait en évaluant l'expression qu'il contient.
— Il ne peut pas y avoir deux fonctions portant le même nom définies dans le même bloc (les fonctions définies avec **`def`** ne peuvent pas être surchargées).

Le programme suivant est un exemple de code ϵ-lang. Il calcule 10!:

```
{
    val goal : int = 10
    def fact(x : int) : int = {
        def inner(i : int) : int = {
            x < i || i * inner(i + 1)
        }
        inner(1)
    }
    fact(goal)
}
```

Exercice 1 : Analyse Lexicale

Décrivez un analyseur lexical qui reconnaisse des séquences comprenant des éléments des classes suivantes de lexèmes de ϵ-lang:

- les identifiants
- les nombres entiers donnés en notation décimale
- les nombres entiers donnés en hexadécimal
- les nombres entiers donnés en octal
- les nombres à virgule

. . . et qui ne tienne pas compte des caractères d'espacement entre les lexèmes. (Repésentez les caractères d'espacement par le symbol $\sqcup$.)

Présentez votre analyseur lexical sous forme d'une machine à états finis comprenant plusieurs états acceptants correspondant aux différentes classes de lexèmes. Prenez garde à correctement implémenter la reconnaissance de chaînes les plus longues possibles (*longest match*).

Vous pouvez étiquetter les transitions de votre automate avec des intervalles de caractères (ex. [a-z]), mais n'utilisez pas d'opérateurs s'appliquant aux expressions régulières tels que `|`, `*` ou `+` (vous devez les encoder en utilisant les états et les transitions appropriées dans votre automate).

Exercice 2 : Analyse Syntaxique

1. Considérez la grammaire de ϵ-lang. Calculez:
   (a) l'ensemble des symboles non-terminaux *nullable* (desquels on peut dériver la chaîne vide)
   (b) l'ensemble $first(\textit{defn})$
   (c) l'ensemble $follow(\textit{block})$
2. Donnez au moins une raison spécifique pour laquelle la grammaire n'appartient pas à la classe $LL(1)$ (indiquez les éléments concernés dans *nullable*, $first$ et $follow$ ainsi que les règles de grammaire concernées).
3. Reformulez la production pour les expressions *ex* en plusieurs règles de grammaire équivalentes de telle sorte que la priorité et l'associativité des opérateurs `*`, `/`, `+`, `<`, || soit correctement encodées et que la récursivité à gauche soit éliminée de la définition de *ex*.
4. En vous basant sur votre nouvelle grammaire, dessiner l'arbre de syntaxe concrète de l'expression:

```
a + y < p / z / w * u
```

5. Notez que les expressions de notre langage peuvent être mises entre parenthèses à souhait et que, pour une fonction f de type:

```
(int) ⇒ ((int) ⇒ int)
```

   ... notre langage nous permet d'écrire une expression telle que `f(x)(y)`, ce qui est équivalent à:

```
{
    val resf : (int) ⇒ int = f(x)
    val res : int = resf(y)
    res
}
```

   Expliquez quels problèmes une telle syntaxe présente pour un parseur $LL(1)$. Proposez un changement simple dans les règles de syntaxe concrète qui résoudrait ceci.

Exercice 3 : Analyse de Types

1. Présentez des règles de typage pour les éléments suivant de ϵ-lang:
   (a) l'opérateur de division (`/`)
   (b) l'opérateur plus-petit-que (`<`)
   (c) l'application de fonctions
   (d) les définitions de fonctions (*fDef*)
   (e) les blocs ({ ... })

   Écrivez toutes vos règles en prenant en compte l'environnement Γ. Vous pouvez étendre Γ avec l'opérateur $\biguplus$ et partir du principe qu'il tiendra compte du masquage des définitions imbriquées correctement en accord avec les règles de portée.
   Par exemple, vous pouvez définir ainsi la règle de typage des définitions de valeurs constants:

   $$\textsc{Value Definition}\ \frac{\Gamma \uplus (x \mapsto T_x) \vdash \{\ ds\ ex\ \} : T}{\Gamma \vdash \{\ \texttt{val}\ x : T_x\ ds\ ex\ \} : T}$$

   ...et partir du principe par la suite que x est disponible et correctement typé dans Γ dans le reste du bloc où il est défini.

   Pour vérifier si une variable est présente dans l'environnement, vous pouvez écrire des règles telles que celle-ci:

   $$\textsc{Constant lookup}\ \frac{(x \mapsto T) \in \Gamma}{\Gamma \vdash x : T}$$

   Note: Vous pouvez partir du principe que les fonctions définies dans un bloc ne sont pas récursives, et que vous ne pouvez appeler une fonction que si elle a été définie avant ou si elle a été passée en paramètre.

2. Considérez le programme ϵ-lang suivant:

```
{
    def twice(f : float ⇒ float) : float ⇒ float = {
        def g(x : float) : float = {
            f(f(x))
        }
        g
    }
    def h(x : float) = 0.5 * x
    twice(h)(8 + 2.3)
}
```

   Appliquez vos règles de typage à l'expression `twice(h)(8 + 2.3)` dans le contexte donné par le programme ci-dessus, en partant du principe que `twice` et `h` sont correctement typés et présents dans l'environnement. Montrez l'entier de la dérivation aboutissant au type final de l'expression.

3. Modifiez la règle pour la vérification du type des blocs de telle manière à ce qu'elle accepte les définitions de fonctions récursives.

Exercice 4 : Génération de Code

Considérez le code suivant:

```
val z : int = ((x < 1) || (1000 < y / x)) + 5
```

1. Présentez une séquence de code pour la machine virtuelle Java telle vous la généreriez pour ce code ϵ-lang.

   Partez du principe que `x` et `y` sont des variables de type `int` disponibles aux positions 1 et 2 respectivement, et que vous désirez enregistrer la valeur de `z` à la position 3.

   Si vous le souhaitez, vous pouvez utilisez les bytecodes abstraits de Cafebabe (comme dans votre projet).

   Utilisez des identifiants symboliques pour signaler les adresses de destinations des sauts.

   Voici quelques bytecodes qui peuvent vous être utile (nous ne prétendons pas qu'ils le sont tous, ni qu'ils sont suffisant pour répondre à la question):

   ```
   ILoad(slot : Int)
   IStore(slot : Int)
   IADD, IAND, IOR, IDIV, IMUL
   If_ICmpLt(target: String)
   If_ICmpGe(target: String)
   ```

2. Considérez la définition suivante:

   ```
   val a : int = 5 + b + (c + 1) / d / 9
   ```

   ... où `b`, `c`, `d` et `e` sont tous de type `int`. Réécrivez cette définition de constante en un bloc équivalent ayant la structure suivante:

   ```
   val a : int = {
     val tmp1 : int = e1
     val tmp2 : int = e2
     val tmp3 : int = e3
     val tmp4 : int = e4
     val tmp5 : int = e5
     tmp5
   }
   ```

   Chaque expression `e1`, `e2`, `e3`, `e4`, `e5` doit avoir la forme p ??? q où ??? dénote un opérateur binaire et p, q dénotent des constantes ou des variables.

3. Pour chaque point du bloc, indiquez lesquelles parmi les variables temporaires { `tmp1`, `tmp2`, `tmp3`, `tmp4`, `tmp5` } sont actives (*live*). Partez du principe que toutes les variables `a`, `b`, `c` et `d` sont actives après le bloc, et qu'aucune variable temporaire n'est active après le bloc.

4. Dessinez un graph dont les 5 sommets sont les variables temporaires et où une arrête indique que deux variables ne peuvent pas être stockées dans le même registre. Quel est le nombre minimal de couleurs qui peut être utilisé pour colorier le graph de telle sorte que deux sommets adjacents n'aient jamais la même couleur?

FIN DES QUESTIONS
