

Final Exam

CIS 756, Fall 1998

December 8, 1998

Read the whole exam first (there are 10 pages) and plan your time. You have 1 hour 48 minutes to complete all the questions. There are a total of 100 points. The exam is open book and open notes, but closed neighbors. Good luck!

Name:

Student-ID:

1. Scanning (20 pts)

Consider scanning literals for representing carbohydrates in chemistry. Such literals consist of one or more *parts*, each consisting of C, H, or O, optionally followed by a decimal integer greater than 1 (it can be greater than 9). You may use D_0 to mean decimal digits 0–9, D_1 to mean 1–9, and D_2 to mean 2–9.¹

(a) (10 pts) Give a regular expression for these carbohydrate literals. You may use the iteration operators $*$ and $+$ and $?$.

(b) (10 pts) Give a DFA that recognizes exactly these literals (no ϵ edges). Don't forget to mark the start state and all accepting states. Make sure that it does not accept the empty string!

¹This is a computer science question, not a chemistry question. In particular, this specification allows bogus chemical formulas such as H_{42} .

2. Parsing, context free grammars (15 pts)

The actual notation for carbohydrates allows subparts to be parenthesized, for example $\text{CH}_3(\text{CH}_2)_{10}\text{COOH}$. Parenthesized subparts need not have a subscript, but if it is present, the subscript must be greater than 1. Parentheses can be nested.

(a) (10 pts) Give an LL(1) grammar for the language of carbohydrates that allows such parenthesization.

(b) (5 pts) Give the (non-abstract) parse tree produced by your grammar for $(\text{CH}_3)_2\text{CO}$.

3. Symbol table (10 pts)

Suppose we want to add function overloading to Tiger with similar semantics as in Java. For example, it should be possible to write

```
function f(i: int)    = 1
function f(s: string) = size(s)
```

If an inner scope redefines `f(int)`, then only the outer `f(int)` should be shadowed. The outer `f(string)` should still be visible.

Describe what modifications to the symbol table are necessary for processing overloaded functions. Your proposed implementation should be efficient, i.e., when traversing a symbol table bucket, the comparison for the symbol table key should not become less efficient.

4. Semantics, intermediate code generation (35 pts)

Consider an exception handling construct for Tiger *similar* to that of Java (which borrowed it from Modula-3 and other languages). Here are BNF grammar productions for the construct:

```
 $\langle exp \rangle ::= \text{try } \langle exp \rangle \langle handler \rangle \text{end}$   
 $\langle handler \rangle ::=$   
 $\langle handler \rangle ::= \text{catch } \langle id \rangle : \langle type \rangle \text{in } \langle exp \rangle \langle handler \rangle$   
 $\langle exp \rangle ::= \text{throw } \langle exp \rangle$ 
```

The **execution semantics** of a **try** expression are as follows. Evaluate the **try** $\langle exp \rangle$. If no exception occurs, then that is the value of the whole expression. If an exception *does occur* in the **try** $\langle exp \rangle$, then the first matching handler (one that matches the type of the expression thrown) is evaluated, with its $\langle id \rangle$ bound to the thrown value for the scope of the handler's $\langle exp \rangle$. The value of the **catch** $\langle exp \rangle$ is then the value of the whole **try** expression. **Ignore the possibility that an $\langle exp \rangle$ may not produce a value.**

For example, with the code

```
let function f() = (throw 5; x := 7)
in
  try
    f()
  catch y: int in x := y
end
end
```

the variable **x** would get the value 5. The assignment **x:=7** never gets executed.

Any exception thrown in evaluation of a **catch** propagates outside the entire **try** expression.

The **throw** expression causes an exception, with the value of its $\langle exp \rangle$.

The **implementation strategy** to use is this: **try** expressions are *handled regions*, and an underlying run-time system package will process execution of **throw** expressions using tables of handled regions and handlers, to be provided by the compiler.

You are given the following Tiger-like intermediate code (tree) *statements* to assist with implementing the **try** expression:

HANDLE(n, s) Define handled region n for evaluating statement s . Here n is a tag associated with the **try** $\langle exp \rangle$ at run time used for looking up handlers (catch blocks).

CATCH(n, t, v) Catch exception of type t occurring in region n , and move thrown value into variable v (v is either a **TEMP** or a **MEM** similar as in **MOVE**).

THROW(e, t) Evaluate expression e of type t and throw it. (The type t of expression e is passed along at runtime for looking up the appropriate handler.)

- (a) (10 pts) Devise a code-generation template for generating intermediate code for **try** expressions. Present it as a list of intermediate code statements with variables representing subparts of the **try** expression (i.e., write down the generated code using variable for subparts, as in **MOVE**(v , $\langle exp \rangle$)).

(b) (10 pts) Give the intermediate code tree corresponding to the following `try` expression.

```
try
  x + 1
catch y: int    in y
catch y: float in x + round(y)
end
```

where `x` is an `int` and `round` is a library function which rounds a `float` to an `int`.

(c) (10 pts) What symbol table actions are required to handle the identifiers defined in the **catch** phrase of exception handlers (i.e., to handle **y** above).

(d) (5 pts) What semantic analysis and type checking (if any) are required for the **try** and **throw** expressions. Account for the possibility that an $\langle exp \rangle$ may not produce a value.

5. Semantic analysis, activation records, intermediate code (10 pts)

Suppose we want to change the parameter passing mechanism in Tiger to call-by-reference. Describe which changes (if any) are necessary to semantic analysis, activation records, and the intermediate representation.

6. Intermediate representation, instruction selection (10 pts)

Given the following Tiger expression

```
if a then b else c
```

where **a** is an integer variable (true if $\neq 0$). Assume all variables are in registers.

- (a) (5 pts) Draw a picture of the IR tree of the above expression, using **Ex**, **Nx**, and **Cx** constructors as appropriate (i.e., what the Tiger compiler would produce).

- (b) (5 pts) Explain how the tree would change if every (sub)expression in Tiger would be translated into an **Ex** tree, i.e., into something that returns a value. For reference, class **Ex** is defined on page 160.