1 A Grammar for Simple Arithmetic Expressions

The following is a grammar for simple arithmetic expressions that involve only natural numbers, addition, and multiplication.

\[
\begin{align*}
\langle \text{digit} \rangle & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \text{number} \rangle & ::= \{(\langle \text{digit} \rangle)\}^+ \\
\langle \text{expression} \rangle & ::= (\langle \text{factor} \rangle) \mid (\langle \text{expression} \rangle) + (\langle \text{expression} \rangle) \\
\langle \text{factor} \rangle & ::= (\langle \text{number} \rangle) \mid (\langle \text{factor} \rangle) \ast (\langle \text{factor} \rangle)
\end{align*}
\]

1. What are the tokens of this language?  

**Solution.** The tokens of the language are digits and the symbols + and \(*\).

2. With this grammar, the expression 1+2+3 has two parse trees:

The existence of two parse trees for the same string shows that the grammar is (circle one)  

(a) ambiguous  
(b) context-free  
(c) context-sensitive  

2 pts
(d) left-associative
(e) right-associative

Solution. (a)

2 Evaluating Simple Arithmetic Expressions

For evaluation of expressions by an interpreter (or for code-generation by a compiler), parse trees are not the most convenient representation, because they still contain information about the original string representation that is unnecessary for computation. The following Oz program is a little expression evaluator that assumes its input is in the form of an abstract syntax tree constructed from Oz tuples.

```oz
fun {Eval X}
  case X
    of int(N) then N
    [] add(X Y) then {Eval X}+{Eval Y}
    [] mul(X Y) then {Eval X}*{Eval Y}
  end
end
```

1. For the first parse tree shown in part 2 of the preceding question, draw the corresponding abstract syntax tree. 4 pts

Solution.

```
add
  \ / \n / \ 
add int
  \ / |
 / \ 3
int int
  | 1 2
```

2. Is the `Eval` function tail-recursive? Explain. 2 pts

Solution. No. In fact, none of the recursive calls is a tail-call.

3 Some

Here is a definition of a function that, given a list Xs and a function P, returns `true` if \( P \ x \) returns `true` for some element \( x \) of \( Xs \) and otherwise it returns `false`. 2
fun {Some Xs P}
case Xs of
    nil then false
    [] X|Xr then
        if {P X} then true else {Some Xr P} end
    $-----------------------------------------
end
end

1. Rewrite the underlined if expression using one of Oz’s boolean operators instead.

Solution.

{P X} orelse {Some Xr P}

2. One of the functions we wrote as a homework exercise was the Member function. {Member Y Xs} returns true if Y occurs in list Xs and otherwise it returns false. Here, instead of writing an independent recursive definition of Member, show how to implement it by passing an appropriate function to Some.

Solution.

fun {Member Y Xs}
    {Some Xs fun {$ X} Y==X end}
end

or

fun {Member Y Xs}
    fun {Check X}
        Y==X
    end
    in {Some Xs Check}
end

3. We know that Oz function definitions and function calls are linguistic abstractions; they are defined by translation into kernel language procedure definitions and procedure calls. Translate the definition of Some into a procedure definition in kernel language syntax. The first part of the translation is given; fill in the rest. (Hint: Be sure your translation reproduces the semantics of the original when there is an error in calling Some, with, say, a number instead of a list.)

Some = proc {$ Xs P B}
    case Xs of
        nil then B=false
end
else case Xs of X|Xr then

Solution.

Some = proc {§ Xs P B}
case Xs of
  nil then B=false
else case Xs of X|Xr then
  local Q in
  {P X Q}
  if Q then B=true else {Some Xr P B} end
end
else raise error("missing else clause") end
end
end

4 An Abstract Machine Execution

The following sequence of abstract machine states is the beginning of an execution of an Oz (kernel language) statement, expressed in a format similar to one we’ve used in several lab exercises. Continue the execution to termination.

local Y in
  local X in
    X=1
    local X in
      X=2
      end
    end
  end
end
Env: {}

Store: {}

⇒
local X in
  X=1
local X in
  X=2
end
Y=X
end

Env:\{Y \rightarrow y\}

Store:\{y\}

\Rightarrow

\begin{array}{l}
X=1 \\
local X in \\
  X=2 \\
end \\
Y=X \\
\end{array}

Env:\{Y \rightarrow y, X \rightarrow x\}

Store:\{y, x\}

\Rightarrow

\begin{array}{l}
X=1 \\
Env:\{Y \rightarrow y, X \rightarrow x\} \\
local X in \\
  X=2 \\
end \\
Y=X \\
Env:\{Y \rightarrow y, X \rightarrow x\} \\
Store:\{y, x\} \\
\end{array}

\Rightarrow

Solution.
\[
\text{local } X \text{ in } \\
\begin{array}{l}
X = 2 \\
\text{end} \\
Y = X
\end{array}
\]

\[
\text{Env: } \{ Y \rightarrow y, X \rightarrow x \}
\]

\[
\text{Store: } \{ y, x = 1 \}
\]

\[
\Rightarrow
\]

\[
\text{local } X \text{ in } \\
\begin{array}{l}
X = 2 \\
\text{end} \\
Y = X
\end{array}
\]

\[
\text{Env: } \{ Y \rightarrow y, X \rightarrow x \}
\]

\[
\text{Store: } \{ y, x = 1 \}
\]

\[
\Rightarrow
\]

\[
X = 2 \\
\text{Env: } \{ Y \rightarrow y, X \rightarrow x' \}
\]

\[
\text{Store: } \{ y, x = 1, x' \}
\]

\[
\Rightarrow
\]

\[
Y = X \\
\text{Env: } \{ Y \rightarrow y, X \rightarrow x \}
\]

\[
\text{Store: } \{ y, x = 1, x' = 2 \}
\]

\[
\Rightarrow
\]
Store: \{y = 1, x = 1, x' = 2\}

Terminates.

Midterm Exam, Part II

5  Arity

What does \{Arity r(b:1 a:2 3:7)\} return? 2 pts

Solution.  3 a b
(2 pts for this exact answer, 1 pt if these features are listed but in a different order)

6  Free and Bound Identifiers

List for each of the following Oz statements the free and bound variable identifiers. (Don’t forget: an identifier can be both free and bound in the same statement, because one occurrence of it is free and another occurrence is bound.) 6 pts

1. \{P X Y\} local X in \{X P Y\} end
   Free: ____________ Bound: ____________

2. local X in local Y in \{X Y Z\} end end
   Free: ____________ Bound: ____________

3. case X of f(Y) then \{P Y\} else \{Q Y\} end
   Free: ____________ Bound: ____________

Solution.

1. Free: P, X, Y; bound: X.

2. Free: Z; bound: X, Y.

3. Free: P, X, Y, Q; bound: Y.
7 Producing Power Functions

Consider the following variant of the Pow-function we’ve seen in lectures and labs.

```
fun {PowFactory X}
  fun {PowAcc N A}
    if N==0 then A
    else {PowAcc N-1 X*A}
  end
end
fun {Pow N}
  {PowAcc N 1}
end
in
  Pow
end
```

1. Write an expression using PowFactory to compute 3 to the 5-th power. 4 pts

**Solution.**

```
local P3 = {PowFactory 3} in {P3 5} end
```

or simply

```
{{PowFactory 3} 5}
```

2. Give the external references for PowAcc 2 pts

**Solution.** PowAcc itself and X.

3. Give the external references for Pow. 2 pts

**Solution.** PowAcc.

8 Semantics of exceptions

When an exception is raised in an Oz program by the (kernel language) semantic statement

```
(raise ⟨x⟩ end, E)
```

semantic statements are popped off the semantic stack looking for a catch statement.
1. No computation is involved in processing each semantic statement that is popped off, other than checking whether or not it is a catch. In C++, on the other hand, the corresponding way of raising an exception (with a throw statement) must process each popped stack frame to call the destructors of all objects referenced in the stack frame. Explain why this is necessary in a C++ program but not in an Oz program.

Solution. The main point to note in the answer is that C++ doesn’t have automatic garbage collection, while Oz does. As discussed in class, since C++ doesn’t have it, the referenced object destructors have to be called since otherwise there could be a memory leak. In Oz, on the other hand, when a store variable is no longer accessible (no environment on the semantic stack has a binding that maps to it or to a record containing it), the garbage collector will reclaim it.

2. Assuming \((\text{catch } \langle y \rangle \text{ then } \langle s \rangle \text{ end, } E_c)\) is the catch statement found, which of the following is the semantic statement pushed on the semantic stack as the last step of processing the raise statement? (Hint: Only one of these really makes any sense, if you remember how environments are used in the abstract machine.)

(a) \((\langle s \rangle, E + \{\langle y \rangle \rightarrow E_c(\langle x \rangle)\})\)
(b) \((\langle s \rangle, E + \{\langle y \rangle \rightarrow E(\langle x \rangle)\})\)
(c) \((\langle s \rangle, E_c + \{\langle y \rangle \rightarrow E(\langle x \rangle)\})\)
(d) \((\langle s \rangle, E_c + \{\langle y \rangle \rightarrow E_c(\langle x \rangle)\})\)

Solution. (c)

9 A Translation Attempt

The semantics of the try-finally statement in Oz

\textbf{try} <s>1 \textbf{finally} <s>2 \textbf{end}

is defined by translating it into a kernel language try-catch:

\textbf{try}
\hspace{1em} <s>1
\textbf{catch} X \textbf{then}
\hspace{1em} <s>2
\hspace{1em} \textbf{raise} X \textbf{end}
\textbf{end}
\hspace{1em} <s>2
If \(<s>2</s>\) is a large statement (perhaps composed of dozens of smaller statements), it’s a problem that this translation duplicates it. Consider the following alternative translation that uses a boolean variable to keep track of whether an exception has occurred in order to avoid duplicating \(<s>2</s>\):

```haskell
local ExceptionOccurred SavedException in
   ExceptionOccurred=true
   try
      \(<s>1</s>\)
      ExceptionOccurred = false
   catch X then
      SavedException=X
   end
   \(<s>2</s>\)
   if ExceptionOccurred then
      raise SavedException end
end
end
```

However, this statement does not have the same semantics as the first translation. Why not? Make your answer precise by describing a case where the two statements would execute differently.

**Solution.** Assume that neither \(<s>1</s>\) nor \(<s>2</s>\) raises an exception. With the first translation, both are executed and no exception is raised. But with the second translation, there are two assignments, with different values, to `ExceptionOccurred`. Since it’s a single assignment variable, the second assignment itself raises an exception, which will be caught by the catch statement and re-raised at the end.