Parallelism

Modern programming languages that can benefit from parallel computing capabilities afforded by multi-core processors will be more efficient and more likely to be used, especially in large software projects.

Our languages can use parallelism particularly in the context of evaluating the expressions in an instance of the \texttt{Rand}s class. In our current implementation, the expressions are evaluated in first-to-last order (or last-to-first, if that’s what you want to do), but they could in fact be evaluated in parallel, one thread per evaluation. A \texttt{Rand}s object is used both in evaluating the RHS expressions in a \texttt{let} expression and in evaluating the actual parameters in a procedure application. In both of these cases, the evaluations can be done in parallel. In the absence of side-effects, parallel evaluation will be unambiguous, leading to improved performance.
Language INFIX

In all of our languages so far, the following primitive operations – addition (+), subtraction (−), multiplication (⋅), and division (/) – have grammar rules that apply these primitives in prefix form, where the operator occurs before the operands. However, most programming languages use infix mathematical notation for these operations, so that instead of writing (as we would in V6, for example)

\[ + (x, \star (4, y)) \]

one would write

\[ x + 4 \star y \]

It turns out that grammar rules that support infix notation are slightly more complicated than the prefix notation we have been using, but not enormously more so. We proceed to illustrate this in our language INFIX.

A naive attempt to define grammar rules that support infix operations might be to replace our PrimappExp grammar rule with something like this:

\[
\text{<exp>:PrimappExp ::= <exp>arg1 <prim> <exp>arg2}
\]

Unfortunately, this won’t even pass the grammar rules checker, since the grammar rule is left recursive – this rule has the nonterminal <exp> on its LHS, and the same nonterminal <exp> appears at the beginning (on the left) of its RHS. Left recursive rules are not allowed in LL1 grammars, and our plcc tool expects only LL1 grammars.

There’s another problem with this approach, called associativity. Even if left recursion weren’t an issue, how do we deal with expressions like this?

\[ 1 - 2 + 3 \]

Should this be interpreted \( \text{arg1} \) being 1 and \( \text{arg2} \) being the expression \( 2 + 3 \) (with the \( <\text{prim}> \) being SUBOP), or should \( \text{arg1} \) be the expression \( 1 - 2 \) with \( \text{arg2} \) being 3 (with the \( <\text{prim}> \) being ADDOP)? Mathematically, these two interpretations are not the same, but both interpretations would satisfy our grammar rule, and the plcc tool would not know which interpretation to choose.
A related problem is called *precedence*, illustrated by the expression

\[ 1+2 \times 3 \]

If the `plcc` tool chose left associativity (which is what it might have done, correctly, in the previous example), this would be interpreted as `arg1` being `1+2` with `arg2` being `3`, but then the result would be interpreted as 9, whereas the correct interpretation would be 7. The problem is that in infix notation, multiplication has a higher precedence than addition.

So our grammar rules will need to take into account associativity and precedence, neither of which is dealt with in the naive grammar rule given above. We can correct this by introducing grammar rules that avoid left recursion and that accommodate precedence. Here is a set of grammar rules for arithmetic expressions that will do the trick. Note that the `LitExp` and `VarExp` rules have been replaced by `LitFactor` and `VarFactor`.

\[
\begin{align*}
\text{<exp>} & \quad ::= \text{<term>} \text{<terms>} \\
\text{<terms>} & \quad ::= \text{<prim0>} \text{<term>} \\
\text{<term>} & \quad ::= \text{<factor>} \text{<factors>} \\
\text{<factors>} & \quad ::= \text{<prim1>} \text{<factor>} \\
\text{<factor>}:\text{LitFactor} & \quad ::= \text{LIT} \\
\text{<factor>}:\text{VarFactor} & \quad ::= \text{VAR} \\
\text{<factor>}:\text{ParenFactor} & \quad ::= \text{LPAREN} \text{<exp>} \text{RPAREN} \\
\text{<factor>}:\text{Prim2Factor} & \quad ::= \text{<prim2>} \text{<factor>} \\
\text{<prim0>}:\text{AddPrim} & \quad ::= \text{ADDOP} \\
\text{<prim0>}:\text{SubPrim} & \quad ::= \text{SUBOP} \\
\text{<prim1>}:\text{MulPrim} & \quad ::= \text{MULOP} \\
\text{<prim1>}:\text{DivPrim} & \quad ::= \text{DIVOP} \\
\text{<prim2>}:\text{UminusPrim} & \quad ::= \text{SUBOP}
\end{align*}
\]
Here is a parse trace of the arithmetic expression ‘1-2+3’:

```
<exp>
 | <term>
 | | <factor>:LitFactor
 | | | LIT "1"
 | <factors>
<terms>
 | <prim0>:SubPrim0
 | | SUBOP "-
 | <term>
 | | <factor>:LitFactor
 | | | LIT "2"
 | | <factors>
 <prim0>:AddPrim0
 | | ADDOP "+"
 | <term>
 | | <factor>:LitFactor
 | | | LIT "3"
 | <factors>
```
Language INFIX (continued)

One final problem with using infix notation for arithmetic expressions is that there is nothing specific
to mark the end of the expression. With prefix notation, the end of a primitive application is always a
right parenthesis, but with infix notation, there is nothing similar. In most cases, it’s easy to identify
the end of an expression. Consider, for example, the following:

```markdown
let
  x = 3+4
in
  x+5
```

The expression 3+4 is ended with the reserved word `in`, but there is nothing to indicate that x+5 is
all that’s there – additional terms might follow on subsequent lines. So for expressions `let` and `if`
that have an `<exp>` at the end, our INFIX grammar rules use end-of-expression markers `end` and
`endif`, respectively. The special token `#` marks the the end of a program, since this symbol is not
used for any other purpose in our language.

Our implementation of the INFIX language is based on the V3 language (with `if` and `let` but no
procs). The semantics of this language is simply to produce the text of the original program in one
line. We relegate the implementation of arithmetic evaluation semantics to exercises.
The ARRAY language extends the OBJ language by adding support for arrays. This language also defines the while primitive. An array of a given size is created using the array operator followed by the size of the array in square brackets. Here's an example:

```plaintext
define a = array[10]
```

When an array is created, its elements are initialized to nil. Array elements are references (in the sense of a ValRef) so they can appear on the LHS of set expressions, and they can refer to any OBJ value, including other arrays. In this way, a two-dimensional array can be constructed as an array of (one-dimensional) arrays.

Array indices are integers that range from zero to the array size minus one. For an array \( a \) and index \( i \), the expression \( a[i] \) refers to the value of the array at the given index. If \( a[i] \) itself refers to an array, the value at its index position \( j \) is \( a[i][j] \).

It is possible to turn an array into a list, and vice versa. For example, here is the code of a procedure \( a2l \) that takes an array parameter and returns a corresponding list. The length of an array \( a \) is written as \( \text{len}(a) \).

```plaintext
% turn an array into a list
define a2l = proc(a)
  let
    i = len(a)
    lst = []
in
  while i do
    { set i = sub1(i)
    ; set lst = addFirst(a[i], lst)
    }
  else
    lst
```

```
Here’s a recursive version of a2l:

% turn an array into a list
define a2l = proc(a)
  let
    alen = len(a)
  in
  letrec
    loop = proc(i)
      if <?(i, alen)
        then addFirst(a[i], .loop(add1(i)))
        else []
      else []
    in
    .loop(0)