COL226 : Programming Languages

Vasu Jain

March 28, 2021

(a) (15 points) Design an implementation in SML, with the help of a stack data structure, for the evaluation as a semantic action for the grammar $G = \langle \{E\}, \{+, *, i\}, \{E \to EE + |EE * |i\}, E\rangle$ where

- $+$ and $*$ are postfix binary operators for addition and multiplication on integers respectively,
- the i token described by the regular expression $0|[1-9][0-9]*$, represents an integer constant.

Note that the result of expression evaluation must be at the top of the stack. The `STACK` signature is as follows:

```
signature STACK =
sig
    type 'a stack
    exception EmptyStack
    val isEmpty : 'a stack -> bool
    val push : ('a * 'a stack) -> 'a stack
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
end
```

Solution.

The grammar described above allows us to express expressions involving additions and multiplications without use of parenthesis due to the postfix nature of the operators.

We shall consider the input as a list of tokens, where i is `NUM i`; $+$ is `PLUS` and $*$ is `MULT`. Thus if the expression is $12 + 3*$ the `SML` equivalent is $[NUM1; NUM2; PLUS; NUM3; MULT]$

```
type token = NUM of int | PLUS | MULT
exception InvalidExpression

fun extractNum = match token with NUM i -> i | _ -> raise InvalidExpression

fun rec evaluate tokenlist numstack =
    match tokenlist with
        | [] -> if isEmpty numstack then raise InvalidExpression else
            extractNum (top numstack)
        | token::tokens -> let updated_numstack =
        ( match token with
            | NUM i -> push i numstack
            | PLUS ->
                let op1 = if isEmpty numstack then InvalidExpression else
                    extractNum (top numstack) and
                let numstack_1 = pop numstack and
                let op2 = if isEmpty numstack_1 then InvalidExpression else
                    extractNum (top numstack_1) and
                let numstack_2 = pop numstack_1 and
                push (op1+op2) numstack_2
            | MULT ->
```

```
                    let op1 = if isEmpty numstack then InvalidExpression else
                        extractNum (top numstack) and
                    let numstack_1 = pop numstack and
                    let op2 = if isEmpty numstack_1 then InvalidExpression else
                        extractNum (top numstack_1) and
                    let numstack_2 = pop numstack_1 and
                    push (op1*op2) numstack_2
            ); evaluate tokens updated_numstack
```

Rubric.

- 2 point for recursion and pattern matching.
- 2 points for i case.
- 5 point for $+$ case.
- 5 point for $*$ case.
- 1 point for Raising exception.

(b) (5 points) Consider the statement: "*Infix operators at the same precedence level can have different associativities.*" Analyse the statement for its correctness with justifications.

Solution.

Operator Associativity comes into play when two infix operators at the same precedence level(like $+$ and $-$; or $\times$ and $\div$) appear in an expression adjacent to each other. In such cases these operators must have the same associativities else compiler is unable to decide the evaluation order of expressions.

An example would be the expression $1 - 2 + 3$.

- If both $+$ and $-$ have Left Associativity then this expression is evaluated as $(1-2)+3 = -1+2 = 1$.
- If both $+$ and $-$ have Right Associativity then this expression is evaluated as $1-(2+3) = 1-5 = -4$.
- If $+$ is right associaitive and $-$ is left associaitive then compiler is unable to decide how to pair the signs in $-2+$ subexpression.

Thus this statement is false.

This statement is false, consider the operators $+$ and $-$ which have the same precedence level in conventional arithmetic and almost all programming languages. These operators cannot have the different associativities since

Rubric.

- 1 point for correctness.
- 3 points for counter-example.
- 1 point for justification.