**Part 1:** Parser

# Description

Given below is the concrete syntax of the project language. The syntax is based on SML. The language is a hybrid of functional and imperative paradigms. The operational semantics are similar to what we have discussed in class. We will discuss implementation of an interpreter in part 2 of the project. For this part, you will make use of the ANTLR/ANTLRWorks tool to generate a pure parser for this language.

|  |  |  |
| --- | --- | --- |
| INPUT | −→ | EXPR ; |
| EXPR | −→ | if EXPR then EXPR else EXPR |
|  | | | let val **id** = EXPR in EXPR end |
|  | | | let fun **id**(**id**)= EXPR in EXPR end |
|  | | | while EXPR do EXPR |
|  | | | { EXPR *(*; EXPR*)\** } |
|  | | | !EXPR |
|  | | | **id** := EXPR |
|  | | | RELEXPR |
| RELEXPR | −→ | ARITHEXPR *(***relop** ARITHEXPR*)?* |
| ARITHEXPR | −→ | TERM *(***addop** TERM*)\** |
| TERM | −→ | FACTOR *(***mulop** FACTOR*)\** |
| FACTOR | −→ | **num** |
|  | | | true |
|  | | | false |
|  | | | **id** |
|  | | | **id** (EXPR) |
|  | | | (EXPR) |

* Note that the meta-notation *(*X*)\** means zero or more repetitions of X and the meta-notation *(*X*)?* means zero or one X.
* Token **num** is any sequence of digits.
* Token **id** is any sequence of letters and digits that begins with a letter.
* All keywords are reserved.
* Token **addop**s are +, -, |. **mulop**s are \*, /, &. **relop**s are <, =. Together, these are referred to as **binaryop**s. These operators follow the usual precedence and associativity rules. The boolean negation operator (!) has a lower precedence!

1

**Part 2: Interpreter**

**Informal Semantics**

* The language is *statically scoped* and *dynamically typed*. Thus, for example, 1+true is a run-time type error.
* Division by zero is considered an evaluation error.
* The condition expression of an if-then-else expression must be a boolean value. Type equivalence of the then-expression and else-expression is not possible under dynamic typing since only one of the branches will get evaluated.
* *Short-circuit* evaluation is used for the logical operators & and |. However, note that unlike, say, Python, the second argument must be a boolean value if evaluated.
* Parameters are *passed by value*.
* Where applicable, arguments are evaluated in the left-to-right order.
* Equality test of boolean expressions is allowed. Equality of functions is meaningless and thus an error. Inequality of booleans is optional.
* The assignment expression is meaningful only if a binding for the corresponding variable already exists (in the current environment). The new value must be compatible with the existing value. The value returned is the value of its right hand side.
* While-expressions are used for side-effects only. The value they return is of no importance. For this implementation, we will have all while-expressions return the value ‘false’.[[1]](#footnote-1)
* For (binary) sequencing, the value returned is the value of the second expression. In general, for a sequence of expressions, the value returned is that of the last expression in the sequence. Of course, we do have to evaluate each expression in a sequence and that too in the given left-to-right order.

1

1. It would be better to have a type ‘void’ and have while-expressions return the only value of this type. This extension can be incorporated easily. However, our choice here will not affect the expressiveness of the language. [↑](#footnote-ref-1)