Tutorial 6 (Solutions)

Computer Language Processing and Compiler Design
(COMP 4901U)

October 11, 2021

Exercise (1): Pratt Parsing

Recall the basic Pratt parsing algorithm shown in the lecture, which is presented below:

```
class Parser(ite: Iterator[Token]):
  var cur: Option[Token] = ite.nextOption
  def consume: Unit = { cur = ite.nextOption }
  def fail(msg: String): Nothing = throw new Exception(msg)
  def skip(tk: Token): Unit =
      if cur != Some(tk) then fail("expected " + tk + ", found " + cur)
      consume

  def expr(prec: Int): Expr = cur match
      case Some(Ident(nme)) =>
        consume; exprCont(Var(nme), prec)
      case Some(OpenParen) =>
        consume; val res = expr(0); skip(CloseParen)
        exprCont(res, prec)
      case _ => fail(rest)
  // Having parsed acc, what to do next at this precedence?
  def exprCont(acc: Expr, prec: Int): Expr = cur match
      case Some(Oper(opStr)) if opPrec(opStr)._1 > prec =>
        consume
        val rhs = expr(opPrec(opStr)._2)
        exprCont(Infix(acc, opStr, rhs), prec)
      case _ => acc
end Parser
```

Note that this parser only supports infix expressions.

Question 1

Provide the parse result of the following input expressions, assuming that `opPrec("++") == (20, 21)`, `opPrec(":+") == (10, 21)`, and `opPrec("+:") == (20, 11)`:

- `a ++ b ++ c`
- `a +: b +: c`
- `a +: b :+ c`
- `a +: b ++ c`

Solution

- `Infix(Infix(Var("a"), "++", Var("b")), "++", Var("c"))`
- `Infix(Var("a"), "+:", Infix(Var("b"), "+:", Var("c")))`
- `Infix(Infix(Var("a"), "+:", Var("b")), ":+", Var("c"))`
- `Infix(Var("a"), "+:", Infix(Var("b"), "++", Var("c")))`

Question 2

Now, consider an extended language which also includes both prefix and postfix operations. The new token and AST definitions are given below:

```
enum Token: // Same as before
  case OpenParen; case CloseParen
  case Ident(name: String)
  case Oper(name: String)

enum Expr:
  case Var(name: String)
  case Infix(lhs: Expr, op: String, rhs: Expr)
  case Prefix(op: String, rhs: Expr) // New
  case Postfix(lhs: Expr, op: String) // New
```

We provide the following three additional functions, which you will use in your parser implementations:

```
def isInfix(opStr: String): Boolean
def isPrefix(opStr: String): Boolean
def isPostfix(opStr: String): Boolean
```

First, update the parser definition shown above to also parse *prefix* operations (postfix operations will be done in the next question). Assume that the right-precedence of prefix operators, which is the only precedence that is relevant for them, is given through the `opPrec` function as well.

Solution

```
class Parser(ite: Iterator[Token]):
  // ... as before

  def expr(prec: Int): Expr = cur match
      case Some(Ident(nme)) =>
        consume; exprCont(Var(nme), prec)
      case Some(OpenParen) =>
        consume; val res = expr(0); skip(CloseParen)
        exprCont(res, prec)
      case Some(Oper(opStr)) if isPrefix(opStr) => // New
        consume; val rhs = expr(opPrec(opStr)._2)
        exprCont(Prefix(opStr, rhs), prec)
      case _ => fail(rest)

  // ... as before
end Parser
```

Question 3

Note that some operators may be valid operators in *both* infix *and* prefix notation, such as the `+` and `-` operators. Does that fact introduce any potential ambiguities in the parsing semantics? Provide the parse results for the following expressions, assuming that `opPrec("+") == opPrec("-") == (5, 6)` and `opPrec("*") == (7, 8)`:

- `+ - + a + - b`

- `a - b - - c`
- `+ a * - b`

Solution

No ambiguities. We simply follow the left- and right-precedences as before.

- `Infix(Prefix("+", Prefix("-", Prefix("+", Var("a")))), "+", Prefix("-", Var("b")))`
- `Infix(Infix(Var("a"), "-", Var("b")), "-", Prefix("-", Var("c")))`
- `Prefix("+", Infix(Var("a"), "*", Prefix("-", Var("b"))))`

Question 4

Now consider the addition of postfix operation support to the parser. Can this combination of fixities be ambiguous, assuming some operators may be any combination of infix, prefix, and postfix, including all at the same time? If so, provide an example of ambiguity, and propose a way of disambiguating these expressions.

Solution

Ambiguity, assuming `+` has all fixities: `a + + b`
can be parsed as either:
`Infix(Postfix(Var("a"), "+"), "+", Var("b"))`
or
`Infix(Var("a"), "+", Prefix("+", Var("b")))`.

We could decide that the former parse should be the valid one, given that `+` is left-associative, so it makes intuitive sense. We would pick the latter for right-associative operators.

Question 5

Update your previous parser definition to also parse postfix operations, assuming that an operator can never be *both* infix *and* postfix.

Make sure to parse low-precedence postfix operators correctly. For instance, assuming postfix `&` has lowest left-precedence and postfix `!` has highest left-precedence, you should parse `a + b ! &` as `(a + (b !)) &`.

Solution

```
class Parser(ite: Iterator[Token]):
  // ... as before

  def exprCont(acc: Expr, prec: Int): Expr = cur match
      case Some(Oper(opStr)) if isInfix(opStr) && opPrec(opStr)._1 > prec =>
        consume
        val rhs = expr(opPrec(opStr)._2)
        exprCont(Infix(acc, opStr, rhs), prec)
      case Some(Oper(opStr)) if isPostfix(opStr) && opPrec(opStr)._1 > prec =>
        consume
        exprCont(Postfix(acc, opStr), prec)
      case _ => acc

end Parser
```

Question 6

Do you see an easy way of supporting the parsing of operators that are both infix and postfix, while retaining linear time complexity?

Solution

No, because locally determining whether we are parsing an infix or postfix operator is not possible, as it would require a unbounded amount of lookahead. This would be easy to implement through backtracking, but backtracking would make the time complexity of the algorithm much worse.

Exercise (2): Name Analysis

As a reminder, there are two main scoping disciplines in use in programming languages:

- *static* scoping, whereby the binding locations of all names are resolved statically; and
- *dynamic* scoping, whereby names are resolved based on the dynamic context of each function call.

Question

Consider the following program. Show what is output with static scoping and with dynamic scoping.

```
object Ex {
  var x = 0
  def foo(): Unit = {
    var x = 1
    bar()
  }
  def bar(): Unit = {
    println(x)
    x += 1
  }
  foo()
  println(x)
}
```

Solution

With static scoping:

```
0
1
```

With dynamic scoping:

```
1
0
```