Tutorial 4 (Solutions)

Computer Language Processing and Compiler Design
(COMP 4901U)

October 4, 2021

Exercise (1): Recursive-Descent Parsing

Consider the following grammar, representing the types *ty* of a functional programming language like Scala:

$$\begin{aligned} ty &\rightarrow ident \mid ident[tyList] \mid ty\ op\ ty \mid \{\, fieldList \,\} \\ tyList &\rightarrow ty,\ tyList \mid \varepsilon \\ op &\rightarrow \Rightarrow \mid \cap \mid \cup \\ fieldList &\rightarrow ident : ty\,;\ fieldList \mid ident : ty \mid \varepsilon \end{aligned}$$

For example, a valid type is $\mathsf{Int} \cup \mathsf{String} \Rightarrow \{\mathsf{x} : \mathsf{Int};\ \mathsf{y} : \mathsf{Double}\}$, representing functions from a union of integers and strings to a structural record type with fields x and y. Here, Int, String, x, and y are all identifiers, matching the *ident* production of the grammar.

Question 1

Show that this grammar is ambiguous by finding at least three examples showing distinct cases where several parse trees might be possible, given an input sequence of tokens.

Solution

- $\mathsf{Int} \Rightarrow \mathsf{Int} \Rightarrow \mathsf{Int}$ as $(\mathsf{Int} \Rightarrow \mathsf{Int}) \Rightarrow \mathsf{Int}$ or $\mathsf{Int} \Rightarrow (\mathsf{Int} \Rightarrow \mathsf{Int})$;
- $\mathsf{Int} \cup \mathsf{Int} \cap \mathsf{Int}$ as $(\mathsf{Int} \cup \mathsf{Int}) \cap \mathsf{Int}$ or $\mathsf{Int} \cup (\mathsf{Int} \cap \mathsf{Int})$.
- etc.

Question 2

Rewrite this grammar in a form that is non-ambiguous, where:

- The precedence of the operators of the language should satisfy:

  $$prec(\cap) > prec(\cup) > prec(\Rightarrow)$$

  meaning that, for example, $A \cup B \cap C$ should parse as $A \cup (B \cap C)$.

- Unions and intersections should be left-associated, while *function types should be right-associated*, meaning that, for example, $A \cup B \cup C$ should parse as $(A \cup B) \cup C$, but $A \Rightarrow B \Rightarrow C$ should parse as $A \Rightarrow (B \Rightarrow C)$.

Hint: try splitting ambiguous productions into several unambiguous helper productions.

Solution

$$\begin{array}{rcl}
\mathit{ty} & \rightarrow & \mathit{unions} \mid \mathit{unions} \Rightarrow \mathit{ty} \\
\mathit{unions} & \rightarrow & \mathit{intersections} \mid \mathit{unions} \cup \mathit{intersections} \\
\mathit{intersections} & \rightarrow & \mathit{atom} \mid \mathit{intersections} \cap \mathit{atom} \\
\mathit{atom} & \rightarrow & \mathit{ident} \mid \mathit{ident}[\mathit{tyList}] \mid \{\, \mathit{fieldList} \,\} \\
\mathit{tyList} & \rightarrow & \ldots \text{(as before)} \ldots \\
\mathit{fieldList} & \rightarrow & \ldots \text{(as before)} \ldots
\end{array}$$

Question 3

Is your grammar left-recursive? As a reminder, left-recursion occurs when at least one rule of the grammar has a right-hand side alternative which can start by the left-hand side of the rule itself, either *directly*, as in $S \rightarrow S\ A\ B$ or *indirectly*, as in $S \rightarrow A\ B\ C$; $B \rightarrow S\ C$ if we assume that A accepts the empty string.

If your grammar is left-recursive, rewrite it in a form that is not left-recursive. Can you do so while preserving the correct associativity of parse trees? If not, explain why, and what one can do, in the parser implementation, to mitigate this.

Solution

We cannot define a non-left-recursive grammar that parses operators into left-recursive trees. Indeed, such left-recursive trees could have left paths of arbitrary lengths, meaning that the production to parse such paths would have to repeat at some point (there are only finitely many productions), so the grammar would have to be left-recursive.

What we can do is construct the parser in a way that it post-processes the parse tree to restore the correct associativity (or restores it on the fly, if parse trees are never represented in memory).

To fix the grammar in our case, the simplest way is to reverse $\mathit{unions} \cup \mathit{intersections}$ and $\mathit{intersections} \cap \mathit{atom}$ into respectively $\mathit{intersections} \cup \mathit{unions}$ and $\mathit{atom} \cap \mathit{intersections}$ in the previous grammar. However, this simple trick is not always possible. The general way of removing direct left-recursion is to split all left-recursive rules into the "heads", which are the first nonterminals they accept besides themselves, and the corresponding "tails" following these nonterminals:

$$
\begin{aligned}
\mathit{ty} &\rightarrow \mathit{unions} \mid \mathit{unions} \Rightarrow \mathit{ty} \\
\mathit{unions} &\rightarrow \mathit{intersections}\ \mathit{moreUnions} \\
\mathit{moreUnions} &\rightarrow \varepsilon \mid \cup\, \mathit{intersections}\ \mathit{moreUnions} \\
\mathit{intersections} &\rightarrow \mathit{atom}\ \mathit{moreIntersections} \\
\mathit{moreIntersections} &\rightarrow \varepsilon \mid \cap\, \mathit{atom}\ \mathit{moreIntersections} \\
\mathit{atom} &\rightarrow \mathit{ident} \mid \mathit{ident}[\mathit{tyList}] \mid \{\, \mathit{fieldList} \,\} \\
\mathit{tyList} &\rightarrow \ldots \text{(as before)} \ldots \\
\mathit{fieldList} &\rightarrow \ldots \text{(as before)} \ldots
\end{aligned}
$$

Question 4

Write a Scala algorithm that can parse your grammar by using *recursive descent*. The principle of a recursive descent parser is very simple: each production of the grammar is associated with a recursive function, which analyses the stream of tokens one by one to decide what to do next, building the abstract syntax tree (AST) as a result. Note that your parser will have to reconstruct ASTs following the correct associativity of operations.

Assume that your algorithm takes as input a List[Token]. As a reminder, List[A] in Scala is defined inductively as either of two constructors $x :: xs$ (with x of type A and xs of type List[A]) or Nil.

Tokens are defined as follows:

```
enum Token:
  case Ident(name: String)
  case Bracket(curly: Boolean, opening: Boolean)
  case Comma
  case Colon
  case Semi
  case Arrow
  case Cup
  case Cap
```

The syntax of the abstract syntax trees you are supposed to output is defined as follows:

```
enum Type:
  case Simple(name: String)
  case Applied(prefix: String, args: List[Type])
  case Infix(lhs: Type, op: Op, rhs: Type)
  case Record(fields: List[(String, Type)])

enum Op { case Fun, Inter, Union }
```

For example, given an input containing the successive values Ident("Int"), Cup, Ident("String"), Cup, Ident("List"), Bracket(**false**, **true**), Ident("Int"), and Bracket(**false**, **false**), which corresponds to the input Int∪String∪List[Int], your parser should return Infix(Infix(Simple("Int"), Union, Simple("String")), Union, Applied("List", Simple("Int") :: Nil)). For input that cannot be parsed according to the grammar, your parser may throw a runtime exception.

Solution

```
def fail(ts: List[Token]): Nothing =
  throw new IllegalArgumentException("Parse failure on: " + ts.mkString("|"))

def parse(ts: List[Token]): Type =
  val (t, rest) = ty(ts); if rest.nonEmpty then fail(rest); t

def ty(ts: List[Token]): (Type, List[Token]) =
  val (us, rest) = unions(ts)
  rest match
    case Arrow :: rest =>
      val (t, rest2) = ty(rest)
      (Infix(us, Fun, t), rest2)
    case _ => (us, rest)

def unions(ts: List[Token]): (Type, List[Token]) =
  val (is, rest) = intersections(ts)
  val (tys, rest2) = unionsList(rest)
  (tys.foldLeft(is)((a, b) => Infix(a, Union, b)), rest2)

def unionsList(ts: List[Token]): (List[Type], List[Token]) = ts match
  case Cup :: rest =>
    val (t, rest2) = intersections(rest)
    val (tys, rest3) = unionsList(rest2)
    (t :: tys, rest3)
  case _ => (Nil, ts)

def intersections(ts: List[Token]): (Type, List[Token]) =
  val (is, rest) = atom(ts)
  val (tys, rest2) = intersectionsList(rest)
  (tys.foldLeft(is)((a, b) => Infix(a, Inter, b)), rest2)

def intersectionsList(ts: List[Token]): (List[Type], List[Token]) = ts match
  case Cap :: rest =>
    val (t, rest2) = atom(rest)
    val (tys, rest3) = intersectionsList(rest2)
    (t :: tys, rest3)
  case _ => (Nil, ts)
```

```
def atom(ts: List[Token]): (Type, List[Token]) = ts match
  case Ident(nme) :: Bracket(false, true) :: rest =>
    val (tys, rest2) = tyList(rest)
    rest2 match
      case Bracket(false, false) :: rest3 => (Applied(nme, tys), rest3)
      case _ => fail(rest2)
  case Bracket(true, true) :: rest =>
    val (tys, rest2) = fieldList(rest)
    rest2 match
      case Semi :: Bracket(true, false) :: rest3 => (Record(tys), rest3)
      case Bracket(true, false) :: rest3 => (Record(tys), rest3)
      case _ => fail(rest2)
  case Ident(nme) :: rest => (Simple(nme), rest)
  case _ => fail(ts)

def tyList(ts: List[Token]): (List[Type], List[Token]) =
  val (t, rest) = atom(ts)
  rest match
    case Colon :: rest2 =>
      val (tys, rest3) = tyList(rest2)
      (t :: tys, rest3)
    case _ => (t :: Nil, rest)

def fieldList(ts: List[Token]): (List[(String, Type)], List[Token]) = ts match
  case Ident(nme) :: Colon :: rest0 =>
    val (t, rest) = ty(rest0)
    rest match
      case Semi :: rest =>
        val (tys, rest2) = fieldList(rest)
        ((nme, t) :: tys, rest2)
      case _ => ((nme, t) :: Nil, rest)
  case _ => (Nil, ts)
```

Question 5

Notice that our type grammar is not quite satisfactory, as it does not allow parenthesizing type subexpressions in order to make them parse differently than implied by the standard operator precedences.

What if we wanted to use curly braces for parenthesization of type ex-

pressions (as well as for structural records)? That is to say, what if the *ty* production was defined as:

$$ty \;\rightarrow\; \ldots(\text{as before})\ldots \;\mid\; \{\, ty \,\}$$

so that it would be possible to write types such as $\{\mathsf{Int} \cup \mathsf{String}\} \cap \mathsf{T}$.

Figure out the changes that would need to be made to your unambiguous non-left-recursive grammar, as well as to your Scala recursive-descent algorithm. Note: think about how to parse things like {{Int}} and {{}}.

Solution

Modifying the grammar is straightforward. As for the implementation, we simply have to perform a little more lookahead, as in:

```
def atom(ts: List[Token]): (Type, List[Token]) = ts match
  case Ident(nme) :: Bracket(false, true) :: rest =>
    ... // as before
  case Bracket(true, true) :: rest => rest match
    case Ident(_) :: Colon :: _ | Bracket(true, false) :: _ => // lookahead
      val (tys, rest2) = fieldList(rest)
      ... // as before
    case _ => // new case
      val (t, rest2) = ty(rest)
      rest2 match
        case Bracket(true, false) :: rest3 => (t, rest3)
        case _ => fail(rest2)
  case Ident(nme) :: rest => (Simple(nme), rest)
  case _ => fail(ts)
```

Exercise (2): Language of Prime Numbers

Let A be the singleton alphabet containing only the symbol 1. Let L be the language of words over A whose size is a prime number:

$$L = \{w \in A^* \mid |w| \text{ is prime}\}$$

Prove that L is regular by building a regular expression for L **or** prove that L is not regular using the *pumping lemma*.

As a reminder, the pumping lemma states that, for any regular language L, there exists a strictly positive constant number p such that every word w in L whose length is at least p can be written as $w = xyz$ where:

1. $|y| > 0$
2. $|xy| \leq p$
3. For any number i, we have that $xy^iz \in L$

Solution

We prove this by contradiction. Assume L is regular.

Let $w \in L$ be a word such that $|w| > 1$ and $|w| > p$, where p is the pumping constant. Such words exist because of the infiniteness of prime numbers.

According to the lemma, $w = xyz$ with $|y| > 0$. Moreover, for any i, we must have that $xy^iz \in L$ and thus that $|xy^iz|$ is prime.

Also, note that $|xy^iz| = |x| + i|y| + |z| = |w| + (i-1)|y|$.

Consider the case $i = |w|+1$. We must have that $|w|+|w|\cdot|y| = |w|(|y|+1)$ is prime, but since $|y| + 1 > 1$ and $|w| > 1$, this number is not prime as it is the product of two natural numbers greater than 1.

$\Rightarrow$ **Contradiction**.