British Informatics Olympiad Final

30 March – 1 April, 2007

Sponsored by Lionhead Studios

Parsing

This question is quite long — do not worry if you do not get to the end

Backus-Naur form

One of the first tasks when writing a compiler for a programming language is to write code to 'understand' the program we want to compile. This process, called *parsing*, breaks down source code into components, often converting them into some kind of intermediate language. The first step is to detail what statements are allowed in the language, and how they are allowed to be built up into programs. *Backus-Naur form* (or BNF) is a convenient way of doing this.

Consider the following BNF description (called a *grammar*) for a language:

```
PET = "dog" | "goldfish"
APOLOGY = "I'm sorry: " | "I apologise but " | "I regret to inform you that "
EXCUSE = "because my hard disk failed." | "because my " & PET & " ate it."
STATEMENT = APOLOGY & "I didn't do my homework " & EXCUSE
```

Each line defines a new element (called a *nonterminal*) which together make up our grammar. In this grammar, the nonterminals are called `PET`, `APOLOGY`, `EXCUSE` and `STATEMENT`. The symbol `|` indicates choice, so the first line says that a `PET` is either the string `"dog"` or the string `"goldfish"`. The symbol `&` denotes joining, so the third statement tells us an excuse is either the string `"because my hard disk failed."` or the string `"because my "`, followed by a `PET`, followed by the string `" ate it."`.

Thus a complete `STATEMENT` in the grammar above might be `"I'm sorry: I didn't do my homework because my goldfish ate it."`

Question 1

How many different strings are valid `STATEMENT`s according to the grammar above?

Question 2

Write a simple grammar which defines a nonterminal `PAIR`, where the strings which are valid `PAIR`s are two-character strings whose first character is one of the first five letters of the alphabet and whose second character is one of the last five. You may define nonterminals other than `PAIR`.

We can make more complicated grammars using BNFs, by allowing some of the nonterminal definitions to be *recursive*; that is to refer to themselves. For example, consider the grammar:

```
BOUTROS = "Boutros " & BOUTROS | "Boutros "
NAME = "Dr " & BOUTROS & "Ghali"
```

Question 3

Give three different examples of strings which are valid `NAME`s according to this grammar. How many different statements are valid in total?

Question 4

Write a simple grammar defining a nonterminal called `WHOLE` where the strings which are valid `WHOLE`s are whole numbers written in decimal notation. For example `43210`, `-123`, `0` and `1984` should be valid, but `0111` should be invalid.

Question 5

A very simple language has functions `add(`a`,`b`)`, `max(`a`,`b`)` and `multiply(`a`,`b`)` (where a and b are whole numbers), and allows you to make simple arithmetic problems, like `add(max(10,-1),100)`. Write a grammar for this language. You may assume you have a pre-built nonterminal called `WHOLE`, which matches strings that are whole numbers written in decimal notation.

Some grammars allow you to build a string in more than one way. For example the string `"axb"` can be built up in multiple ways from the grammar:

```
CODE = "x" | "a" & CODE | CODE & "b"
```

Question 6
What are these two different ways?

Question 7
Suppose we have a 7 character string which is a valid `CODE`. What is the maximum number of ways it could be formed? What is the minimum?

Recursive descent parsers

One type of parser is called a *recursive descent* parser. Initially, we will make very simple recursive descent parsers whose only purpose is to tell you whether the input string is valid according to the grammar. We will illustrate a way of making such parsers, which we will call the *RDP technique*, using the original `STATEMENT` grammar which we considered before.

First, suppose that we have a subroutine called `match` which takes an integer p and a string S, and which has access to the input we are trying to parse. If the characters in the input, starting from position p, match S, then the routine returns $p + n$ (where n is the length of S). If it tries to read off the end of the input, or does not match, it always returns -1. Thus if the input was `ABBA`, `match(1,"AB")` would be `3`, `match(2,"BB")` would be `4`, and `match(2,"AB")` would be `-1`.

Then, we go through the grammar in order. For each nonterminal defined in the grammar, we mechanically write a subroutine. This subroutine contains one section for each of the options (parts between `|`s) on the right hand side of the = sign for that nonterminal. Here are the subroutines for our `STATEMENT` grammar:

```
parsePET(p)
{
  try1 ← match(p, "dog")          assign the result of the match to the variable try1
  if (try1 ≠ -1) then return try1 if successful, return success from parsePET

  try2 ← match(p, "goldfish")     assign the result of the match to the variable try2
  if (try2 ≠ -1) then return try2 if successful, return success from parsePET

  return -1                       return failure to match
}                                 end of parsePET

parseAPOLOGY(p)
{
  try1 ← match(p, "I'm sorry:  ")                  assign to the variable try1
  if (try1 ≠ -1) then return try1                  if OK, return success

  try2 ← match(p, "I apologise but ")              etc.
  if (try2 ≠ -1) then return try2

  try3 ← match(p, "I regret to inform you that ")
  if (try3 ≠ -1) then return try3

  return -1
}
```

```
parseEXCUSE(p)
{
  try1 ← match(p, "because my hard disk failed.")
  if (try1 ≠ -1) then return try1

  try2 ← match(p, "because my ")
  if (try2 ≠ -1) then                    if successful, execute block 1
  {                                      start of block 1
    try2b ← parsePET(try2)               assign to the variable try2b
    if (try2b ≠ -1) then                 if successful, execute block 2
    {                                    start of block 2
      try2c ← match(try2b, " ate it.")
      if (try2c ≠ -1) then return try2c
    }                                    end of block 2
  }                                      end of block 1

  return -1
}
parseSTATEMENT(p)
{
  try1 ← parseAPOLOGY(p)
  if (try1 ≠ -1) then
  {
    try1b ← match(try1, "I didn't do my homework ")
    if (try1b ≠ -1) then
    {
      try1c ← parseEXCUSE(try1b)
      if (try1c ≠ -1) then return try1c
    }
  }

  return -1
}
```

Question 8

We parse a `STATEMENT` by calculating `parseSTATEMENT(1)` and looking at the result. If the result is the length of the input string plus 1 we have a valid string. What could be returned if the input was invalid?

Question 9

Use the RDP technique to write pseudocode to parse the grammar you wrote in question 5. You may write in any style of pseudocode you like; you do not need to match the examples above.

Question 10

Of course, a program that just tells us whether the input is valid according to the grammar, but does not do anything else with it, is not very helpful. However, once we have such a program, we can easily modify it to do more exciting things. For instance, how would you modify the pseudocode in question 9 to actually compute the answer to the sum, rather than just saying whether the input is valid? You need not write out the modified pseudocode in full.

The RDP technique does not always give a parser which operates correctly. Most of the time, you can fix this by changing the BNF you use. For instance, consider the following four grammars, all of which allow the same valid strings:

```
BOUTROS = "Boutros " & BOUTROS | "Boutros "
NAME = "Dr " & BOUTROS & "Ghali"
```

```
BOUTROS = "Boutros " | "Boutros " & BOUTROS
NAME = "Dr " & BOUTROS & "Ghali"
```

```
BOUTROS = BOUTROS & "Boutros " | "Boutros "
NAME = "Dr " & BOUTROS & "Ghali"
```

```
BOUTROS = "Boutros " | BOUTROS & "Boutros "
NAME = "Dr " & BOUTROS & "Ghali"
```

Question 11

If we apply the RDP technique above to write pseudocode to parse each of these grammars, we only get a program which works correctly in some cases. Which ones work? What goes wrong in the other cases?

Romulus and Remus are trying to make a simple calculator program, which will work out the answers to simple arithmetic problems like `2+3*4`, where all the numbers involved are one digit long. Romulus proposes to solve the problem by taking the following grammar:

```
NUMERAL = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
TERM = NUMERAL | "(" & FORMULA & ")"
PROD = TERM & "*" & PROD | TERM
FORMULA = PROD & "+" & FORMULA | PROD
```

using it to create a recursive descent parser using the RDP technique (as in question 9), then converting it to give a numerical answer (as in question 10). Remus says that this is much more complicated than it needs to be. He proposes doing the same starting from the following simpler grammar:

```
NUMERAL = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
FORMULA = "(" & FORMULA & ")" | NUMERAL & "+" & FORMULA |
                                        NUMERAL & "*" & FORMULA | NUMERAL
```

which, he says, has the same collection of valid strings. Romulus demurs, claiming that this approach will give the wrong answer for `3+3*3+3`.

Question 12

Is Remus right that the two grammars have the same collection of valid strings? Who is right on the question of whether they need to use the more complicated grammar or whether the simpler grammar will suffice? Justify your answer.

Table-driven LR(0) parsers

Another kind of common parser is a *table-driven LR(0) parser*. These are able to parse fewer languages than recursive descent parsers, but they are much faster when they work. To make a table-driven LR(0) parser for a language, we first run the BNF for the grammar through a magic program which creates something called the *parser table*. For instance, given the simple grammar:

```
NUMERAL = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
EXPR = NUMERAL | "(" & EXPR & "+" & EXPR & ")" | "(" & EXPR & "*" & EXPR & ")"
```

It might produce a table like the following, which will be explained later:

Number	Symbol	Instructions
[1]	`NUMERAL`	**reduce:** `NUMERAL` → `EXPR`
	`EXPR`	**done**
	`"("`	**goto:** [2]
	`"1"`	**reduce:** `"1"` → `NUMERAL`
	`"2"`	**reduce:** `"2"` → `NUMERAL`
	`"3"`	**reduce:** `"3"` → `NUMERAL`
	`"4"`	**reduce:** `"4"` → `NUMERAL`
	etc.	(similar lines for 5,6,7,8,9,0)
[2]	`"("`	**goto:** [2]
	`EXPR`	**goto:** [3]
	`NUMERAL`	**reduce:** `NUMERAL` → `EXPR`
	`"1"`	**reduce:** `"1"` → `NUMERAL`
	`"2"`	**reduce:** `"2"` → `NUMERAL`
	etc.	(similar lines for 3,4,5,6,7,8,9,0)
[3]	`"+"`	**goto:** [4]
	`"*"`	**goto:** [5]
[4]	`"("`	**goto:** [2]
	`EXPR`	**goto:** [6]
	`NUMERAL`	**reduce:** `NUMERAL` → `EXPR`
	`"1"`	**reduce:** `"1"` → `NUMERAL`
	`"2"`	**reduce:** `"2"` → `NUMERAL`
	etc.	(similar lines for 3,4,5,6,7,8,9,0)
[5]	`"("`	**goto:** [2]
	`EXPR`	**goto:** [7]
	`NUMERAL`	**reduce:** `NUMERAL` → `EXPR`
	`"1"`	**reduce:** `"1"` → `NUMERAL`
	`"2"`	**reduce:** `"2"` → `NUMERAL`
	etc.	(similar lines for 3,4,5,6,7,8,9,0)
[6]	`")"`	**reduce:** `"(" & EXPR & "+" & EXPR & ")"` → `EXPR`
[7]	`")"`	**reduce:** `"(" & EXPR & "*" & EXPR & ")"` → `EXPR`

The parser then reads through the input, character by character. At all times, it maintains a special list called the *stack*, which contains two kinds of things: numbers (which we will write in square brackets for clarity) and *symbols*. A symbol is either a character or the name of a nonterminal: for example, `"1"`, `"2"` or `NUMERAL`. The list always alternates between numbers and symbols. When we start, the list contains only one element: the number [1].

We then repeatedly perform the following procedure. If the last thing in the list is a number, we read a character from the input, and put it at the end of the list. If the last thing on the list is a symbol, we look at the second to last thing in the list (which will be a number), and the last thing on the list (which will be a symbol), and we find the row in the parse table corresponding to that number and symbol. We then carry out the instruction in that row or, if there is no such row, report that the input was not valid. There are three possible instructions we might be asked to carry out:

- If the instructions say **goto:** followed by a number, we put that number at the end of the list.
- If the instructions say **reduce:** followed by something like `"(" & EXPR & "+" & EXPR & ")"` → `NUMERAL`, we look at the last few elements in the list. They should say `"(", EXPR, "+", EXPR, ")"` (or whatever appears to the left of the →), with some numbers in between the symbols. We remove all those symbols (and the numbers in between them), and replace them with the single symbol `NUMERAL` (or whatever appears to the right of the →).
- If the instructions say **done**, we should be at the end of the input. If we are not, we report an error. Otherwise, we have finished.

For example, this table shows the result of applying the parsing algorithm to the string `(1+1)`:

List contents before step	**What happens**
[1]	we read `"("` from the input.
[1],`"("`	we **goto** [2].
[1],`"("`,[2]	we read `"1"` from the input.
[1],`"("`,[2],`"1"`	we **reduce:** `"1"` → `NUMERAL`
[1],`"("`,[2],`NUMERAL`	we **reduce:** `NUMERAL` → `EXPR`
[1],`"("`,[2],`EXPR`	we **goto** [3].
[1],`"("`,[2],`EXPR`,[3]	we read `"+"` from the input.
[1],`"("`,[2],`EXPR`,[3],`"+"`	we **goto** [4].
[1],`"("`,[2],`EXPR`,[3],`"+"`, [4]	we read `"1"` from the input.
[1],`"("`,[2],`EXPR`,[3],`"+"`, [4],`"1"`	we **reduce:** `"1"` → `NUMERAL`
[1],`"("`,[2],`EXPR`,[3],`"+"`, [4],`NUMERAL`	we **reduce:** `NUMERAL` → `EXPR`
[1],`"("`,[2],`EXPR`,[3],`"+"`, [4],`EXPR`	we **goto** [6].
[1],`"("`,[2],`EXPR`,[3],`"+"`, [4],`EXPR`, [6]	we read `")"` from the input.
[1],`"("`,[2],`EXPR`,[3],`"+"`, [4],`EXPR`, [6], `")"`	we **reduce:** `"(", EXPR, "+", EXPR, ")"` → `EXPR`
[1], `EXPR`	we are done

Question 13

Produce a similar table showing the result of applying the algorithm to the string `((1+1)*2)`. You only need to show the first column ('list contents before step') and not the second. If part of one line in the table is the same as the line before, you may simply put "—" in that space.

Question 14

Why do you think that table-driven LR(0) parsers are faster than recursive descent parsers?

Question 15

How would you modify the table-driven LR(0) parser in question 14 to actually compute the answer to the sum, rather than just saying whether the input is valid?

Question 16

Let us now turn to the question of how the magic program creates the instruction table given the input and the output. Consider your grammar from question 5. Work out a table-driven LR(0) parser table for that language.

Some grammars can cause problems for the table-driven LR(0) approach. Consider the grammar:

```
ABC = "a" | "b" | "c"
CDE = "c" | "d" | "e"
WORD = ABC & "x" | CDE & "y"
```

Question 17

What happens if we try to make a table-driven LR(0) parser to parse this language? Would a recursive descent parser work for this grammar?