Computer Science 3 - 2016

Programming Language Translation

Practical for Week 4, beginning 8 August 2016 - Solutions

This practical was not always well done. Many people could "guess" the answers, but could not or did not justify their conclusions. **If these sorts of questions are set in exams it is important to justify your claims, so as not to be seen to be guessing.**

As usual, you can find a "solution kit" as `PRAC4A.ZIP` if you wish to experiment further.

Task 2 - Describe BNF

The development of an EBNF description of BNF is shown below. There are a few tricks to be learned from this one. Firstly, productions are separated one from the next by the end of line, not by a period. This means that we cannot IGNORE the line break characters. This has to be done in a way that depends on your operating system, in general. In practice, we have little alternative but to define the `eol` "character set" as the singleton `CHR(10)` and then define an `EOL` "token" as a single character token, as in the code below. Internally Coco/R has been arranged to map line breaks demarcated by `lf (CHR(10))` as in Unix, or by `cr + lf` (as in WinTel) or even `cr` only (as on Apple computers) to a consistent `lf`. Secondly, we wish spaces to become significant characters within the `nonTerminal` tokens that are demarcated by < > brackets. Thirdly, we do *not* want to define the `terminal` token to include spaces within it, as we need to be able to distinguish each terminal from the next if and when they are separated by spaces. In Cocol there is an implicit `IGNORE CHR(32)` - but this relates to ignoring spaces *between* tokens, as is common in almost all programming languages. The only way we can make spaces significant *within* a token definition is to define the singleton character set `space` to consist of `CHR(32)`, as Coco also forbids you from writing a string into a Cocol definition with an embedded spaces, as exemplified by `"this has some spaces"`. Lastly, BNF notation still allows for the use of (round) parentheses (it is tempting to discard these as well).

Incidentally, spaces are very rarely significant in computer languages - the definition of `nonTerminal` here is one of the very few exceptions one can think of (other than the obvious "string with spaces").

A simple definition of the possible tokens looks like this:

```
CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  alpha  = letter + "0123456789_" .
  lf     = CHR(10) .
  space  = CHR(32) .

TOKENS
  EOL         = lf .
  nonTerminal = "<" { space } letter { alpha | space } ">" .
  terminal    =  letter { alpha } .
```

However, this is not really adequate. We should be able to allow almost anything as a "terminal". But a definition like this is doomed to failure:

```
CHARACTERS
  lf          = CHR(10) .
  space       = CHR(32) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  alpha       = letter + "0123456789_" .
  inTerm      = ANY - CHR(0) .. CHR(32) .

TOKENS
  EOL         = lf .
  nonTerminal = "<" { space } letter { alpha | space } ">" .
  terminal    = inTerm { inTerm } .
```

because one cannot then distinguish terminals from non-terminals (why not?). We might try

```
CHARACTERS
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  alpha       = letter + "0123456789_" .
  graphic     = ANY - control - " " .
  startTerm   = graphic - "<" .
  lf          = CHR(10) .
  space       = CHR(32) .

TOKENS
  EOL         = lf .
  nonTerminal = "<" { space } letter { alpha | space } ">" .
  terminal    = startTerm { graphic } .
```

but this is also inadequate, as a sequence like `(oneTerm|twoTerm)` with no helpful spaces will all be scanned as a single terminal, and furthermore there is no way to represent the metasymbol < as a terminal in a set of productions (or any of the other metasymbols, for that matter). Perhaps the merits of the Wirth/Cocol notation are now becoming more apparent! One could, of course, try to insist that users insert space around all terminals, but to be more helpful it may be best to exclude all the meta-characters from starting a terminal, and then to insist that if one wants them as terminals one should use a `'string'` notation after all. So a complete grammar might reads like this, although this allows one to have multiple and potentially misleading *eps* in a Term, as in

```
<A> ::= a eps eps eps eps b
```

```
COMPILER BNF1 $CN
/* Grammar to describe BNF productions
   P.D. Terry, Rhodes University, 2016 */

CHARACTERS
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  alpha       = letter + "0123456789_" .
  graphic     = ANY - control - " " .
  noQuote     = graphic - "'" .
  startTerm   = graphic - "()<|:'" .
  lf          = CHR(10) .
  space       = CHR(32) .

TOKENS
  EOL         = lf .
  nonTerminal = "<" { space } letter { alpha | space } ">" .
  terminal    = startTerm { graphic } | "'" noQuote { noQuote | "''" } "'" .

COMMENTS FROM "(*" TO "*)"  NESTED
COMMENTS FROM "/*" TO "*/"  NESTED

IGNORE control - lf

PRODUCTIONS
  BNF1       = { Production } EOF .
  Production = nonTerminal "::=" Expression SYNC EOL .
  Expression = Term { "|" Term } .
  Term       = Factor { Factor } .
  Factor     = nonTerminal | terminal | "(" Expression ")" | "eps" .
END BNF1.
```

A slightly better solution is to have the PRODUCTIONS section reading:

```
PRODUCTIONS
  BNF2       = { Production } EOF .
  Production = nonTerminal "::=" Expression SYNC EOL .
  Expression = Term { "|" Term } .
  Term       = Factor { Factor } | "eps" .
  Factor     = nonTerminal | terminal | "(" Expression ")" .
END BNF2.
```

although that still allows one to have multiple and potentially misleading

```
| eps | eps | eps
```

options in an *Expression*. If one want to restrict the right hand side to contain at most one *eps*, and still to have an LL(1) grammar, one is forced to demand that the *eps* appears first, as in the grammar below:

```
PRODUCTIONS
  BNF3       = { EOL } { Production } EOF .
  Production = nonTerminal "::=" Expression EOL { EOL }.
  Expression = [ "eps" "|" ] Term { "|" Term } .
  Term       = Factor { Factor } .
  Factor     = nonTerminal | terminal | "(" Expression ")" .
END BNF.
```

Notice that this last grammar also allows you to have blank lines between productions, which the others do not - do you see why?

Task 3 - Let's take a trip to the C

The Parva grammar you met last week incorporated the following productions to describe the syntax of an *Expression*. This uses the hierarchy of operator precedence that is used in Wirth's languages Pascal, Modula-2 and Oberon.

```
Expression       = AddExp [ RelOp AddExp ] .
AddExp           = [ "+" | "-" ] Term { AddOp Term } .
Term             = Factor { MulOp Factor } .
Factor           =   Designator | Constant
                   | "new" BasicType "[" Expression "]"
                   | "!" Factor | "(" Expression ")" .
AddOp            = "+" | "-" | "||" .
MulOp            = "*" | "/" | "&&" | "%" .
RelOp            = "==" | "!=" | "<" | "<=" | ">" | ">=" .
```

and you were asked to explore the development of a set of productions that would correspond to the hierarchy used in the C family (and some others, incidentally.

Far more "levels" are needed: the answer is

```
Expression       = AndExp { "||" AndExp } .
AndExp           = EqlExp { "&&" EqlExp } .
EqlExp           = RelExp { EqlOp RelExp } .
RelExp           = AddExp [ RelOp AddExp ] .  // note [ ] not { }
AddExp           = MulExp { AddOp MulExp } .
MulExp           = Factor { MulOp Factor } .
Factor           = Primary | ( "+" | "-" | "!" ) Factor .
Primary          =   Designator
                   | Constant
                   | "new" BasicType "[" Expression "]"
                   | "(" Expression ")" .
AddOp            = "+" | "-" .
MulOp            = "*" | "/" | "%" .
EqlOp            = "==" | "!=" .
RelOp            = "<" | "<=" | ">" | ">=" .
```

The grammars are not equivalent. All of the statements below would be regarded as correct by the C grammar. Statements 3 and 5 would be accepted correctly by the Pascal grammar. Statement 2 would be accepted by the system you wrote (but it is really incorrect and you might like to ponder the significance of this. Statements 1 and 4 would, correctly, be rejected by the Pascal grammar. Does that help you see why the grammars are not equivalent?

```
int i, j;
bool b;
b = i > j   ||  j > 4;          // 1
b = (i > j) ||  j > 4;          // 2
b = (i > j) || (j > 4);         // 3
j = + - j;                      // 4
b = ! ! b;                      // 5
```

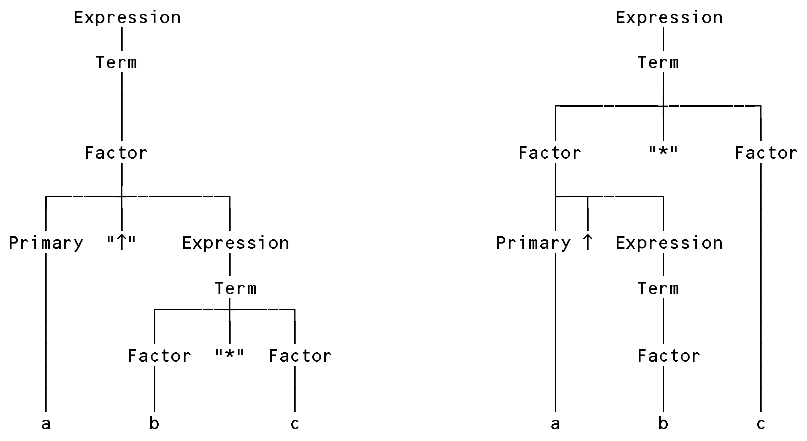
Task 4 - Expressions - again

The following grammar attempts to describe expressions incorporating the familiar operators with their correct precedence and associativity.

```
    COMPILER Expression $CNF
    IGNORE CHR(0) .. CHR(31)
    PRODUCTIONS
      Expression = Term    { ( "+" | "-" ) Term  } .
      Term       = Factor  { ( "*" | "/" ) Factor } .
      Factor     = Primary [ "↑" Expression ] .
      Primary    = "a" | "b" | "c" .
   END Expression.
```

Is this an ambiguous grammar? (Hint: try to find an expression that can be parsed in more than one way).

Again, many people "guessed" the right answer. To justify the claim that it is ambiguous it would be as well to show a pair of parse trees, not just make a wild claim! Considering the expression `a↑b*c`. This can indeed be parsed in two ways, one with the implicit meaning of `a↑(b*c)` and the other with the meaning of `(a↑b)*c`. The parse trees would look like this (a few intermediate nodes have been omitted to save space)



Is it an LL(1) grammar? If not, why not, and can you find a suitable grammar that *is* LL(1)?

It cannot be an LL(1) grammar if it is ambiguous, but let us see which rules are broken. If we rewrite the first grammar to eliminate the metabrackets we get

```
    Expression = Term TailExp .
    TailExp    = AddOp Term TailExp | ε .
    Term       = Factor TailTerm .
    TailTerm   = MulOp Factor TailTerm | ε .
    Factor     = Primary TailFactor .
    TailFactor = "↑" Expression | ε .
    Primary    = "a" | "b" | "c" .
    AddOp      = "+" | "-" .
    MulOp      = "*" | "/" .
```

The nullable nonterminals here are `TailExp`, `TailTerm` and `TailFactor`.

```
    FIRST(TailExp)    = { "+" , "-" }
    FIRST(TailTerm)   = { "*" , "/" }
    FIRST(TailFactor) = { "↑" }
```

The `FOLLOW` sets are a little harder to see because to get to closure one has to chase through quite a few other productions:

```
    FOLLOW(TailExp)    = FOLLOW(Expression)
    FOLLOW(TailTerm)   = FOLLOW(Term) = FIRST(TailExp) U FOLLOW(Expression)
    FOLLOW(TailFactor) = FOLLOW(Factor) = FIRST(TailTerm) U FOLLOW(Term)
```

You are invited to track these through in detail; the outcome is that they are all the same:

```
    FOLLOW(TailExp)    = { "*" , "/" , "+" , "-" , EOF }
    FOLLOW(TailTerm)   = { "*" , "/" , "+" , "-" , EOF }
    FOLLOW(TailFactor) = { "*" , "/" , "+" , "-" , EOF }
```

and so Rule 2 is broken for `TailExp` and for `TailTerm`.

Finding an LL(1), unambiguous grammar, with the correct precedence and associativity is not too difficult. **In fact it would have been incredibly easy had you just read the text, page 67, where the solution is effectively given to you.**

```
    COMPILER Expression $CNF
    IGNORE CHR(0) .. CHR(31)
    PRODUCTIONS
      Expression = Term    { ( "+" | "-" ) Term  } .
      Term       = Factor  { ( "*" | "/*" ) Factor } .
      Factor     = Primary [ "↑" Factor ] .
      Primary    = "a" | "b" | "c" .
   END Expression.
```

Why does our grammar now satisfy the LL(1) constraints? Rewritten it becomes

```
      Expression = Term TailExp .
      TailExp    = AddOp Term TailExp | ε .
      Term       = Factor TailTerm .
      TailTerm   = MulOp Factor TailTerm | ε .
      Factor     = Primary TailFactor .
      TailFactor = "↑" Factor | ε .
      Primary    = "a" | "b" | "c" .
      AddOp      = "+" | "-" .
      MulOp      = "*" | "/" .
```

The nullable nonterminals here are still `TailExp`, `TailTerm` and `TailFactor`.

```
    FIRST(TailExp)    = { "+" , "-" }
    FIRST(TailTerm)   = { "*" , "/" }
    FIRST(TailFactor) = { "↑" }
```

The `FOLLOW` sets are a little harder to see because to get to closure one has to chase through quite a few other productions:

```
    FOLLOW(TailExp)    = FOLLOW(Expression)
    FOLLOW(TailTerm)   = FOLLOW(Term) = FIRST(TailExp) U FOLLOW(Expression)
    FOLLOW(TailFactor) = FOLLOW(Factor) = FIRST(TailTerm) U FOLLOW(Term)
```

You are invited to track these through in detail; the outcome is:

```
    FOLLOW(TailExp)    = { EOF }
    FOLLOW(TailTerm)   = { "+" , "-" , EOF }
    FOLLOW(TailFactor) = { "*" , "/" , "+" , "-" , EOF }
```

and so Rule 2 is no longer broken.

There were various other suggestions made, such as

```
      Factor     = Primary [ "↑" Term ] .
      Factor     = Primary [ "↑" "(" Expression ")" ] .
      Factor     = Primary { "↑" Term } .
```

but these are unnecessarily restrictive (first suggestion) or non-equivalent (second suggestion; parentheses were not catered for in the first grammar and introducing them is "cheating"). The third suggestion gets the associativity incorrect.

Task 5 - Meet the family

Consider the following grammar:

```
COMPILER Home
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS
  Home      = Family { Pets } [ Vehicle ] "house" .
  Pets      = "dog" [ "cat" ] | "cat" .
  Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
  Family    = Parents { Children } .
  Parents   = [ "Dad" ] [ "Mom" ] | "Mom" "Dad" .
  Child     =   "Helen" | "Margaret" | "Alice" | "Robyn" | "Cathy"
              | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
END Home.
```

Analyse this grammar in detail.

The first point to be made is that this is not a reduced grammar. The non-terminal `Child` is unreachable, and there is no way that the non-terminal `Children` can be derived to anything, let alone to terminals. Presumably what was meant was

```
COMPILER Home
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS
  Home      = Family { Pets } [ Vehicle ] "house" .
  Pets      = "dog" [ "cat" ] | "cat" .
  Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
  Family    = Parents { Child } .
  Parents   = [ "Dad" ] [ "Mom" ] | "Mom" "Dad" .
  Child     =   "Helen" | "Margaret" | "Alice" | "Robyn" | "Cathy"
              | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
END Home.
```

If we introduce extra non-terminals to eliminate the [ ] and { } metabrackets we might get:

```
COMPILER Home
IGNORE CHR(0) .. CHR(31)
PRODUCTIONS
  Home        = Family AllPets Vehicle "house" .
  AllPets     = Pets AllPets | .
  Pets        = "dog" OptionalCat | "cat" .
  OptionalCat = "cat" | .
  Vehicle     = TwoWheeled "fourbyfour" | .
  TwoWheeled  = "scooter" | "bicycle" .
  Family      = Parents Children .
  Children    = Child Children | .
  Parents     = OptionalDad OptionalMom | "Mom" "Dad".
  OptionalDad = "Dad" | .
  OptionalMom = "Mom" | .
  Child       =   "Helen" | "Margaret" | "Alice" | "Robyn" | "Cathy"
                | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
END Home.
```

It should be pretty apparent that the productions for `Home` and `Family` cause no problems (no alternatives appear in their right hand sides), nor do the productions for `Pets,` `TwoWheeled` and `Child` (they are not nullable, and the alternatives begin with clearly distinct terminals).

The production for `Parents` needs closer scrutiny.

```
FIRST(Parents_1) = FIRST(OptionalDad) U FIRST(OptionalMom) = { "Dad", "Mom" }
(because OptionalDad is nullable)

FIRST(Parents_2) = { "Mom" }
```

so Rule 1 is broken, and the grammar is not LL(1) compliant.

We can check Rule 2, as there are several productions that have alternatives, one of which is nullable. These are the productions for `AllPets, OptionalCat, Vehicle, Children, Parents, OptionalDad and OptionalMom` (yes, Sylvia, real grammars often have lots of exciting complications).

This means that we must look at

```
FIRST(AllPets) and FOLLOW(AllPets)
FIRST(OptionalCat) and FOLLOW(OptionalCat)
FIRST(Vehicle) and FOLLOW(Vehicle) etc.
```

The results follow

```
FIRST(AllPets)  = { "dog", "cat" }
FOLLOW(AllPets) = { "house", "scooter", "bike" }

FIRST(OptionalCat) = { "cat" }
FOLLOW(OptionalCat) = { "dog", "cat", "house", "scooter", "bike" }
```

(so Rule 2 is broken here, perhaps surprisingly)

```
FIRST(Vehicle)      = { "scooter", "bicycle" }
FOLLOW(Vehicle)     = { "house" }

FIRST(Children)     = { "Helen", "Margaret", "Alice" .... "Ntombizanele" }
FOLLOW(Children)    = { "dog", "cat", "house", "scooter", "bike" }

FIRST(Parents)      = { "Mom", "Dad" }
FOLLOW(Parents)     = { "Helen", "Margaret", "Alice" .... "Ntombizanele",
                        "dog", "cat", "house", "scooter", "bike" }

FIRST(OptionalDad)  = { "Dad" }
FOLLOW(OptionalDad) = { "Mom, "Helen", "Margaret", "Alice" .... "Ntombizanele",
                        "dog", "cat", "house", "scooter", "bike" }

FIRST(OptionalMom)  = { "Mom" }
FOLLOW(OptionalMom) = { "Helen", "Margaret", "Alice" .... "Ntombizanele",
                        "dog", "cat", "house", "scooter", "bike" }
```

We can get an LL(1) description of the family as follows:

```
Home5     = Family { Pets } [ Vehicle ] "house" .
Pets      = "dog" | "cat" .
Vehicle   = ( "scooter" | "bicycle" ) "fourbyfour" .
Family    = Parents { Child } .
Parents   = [ "Dad" [ "Mom" ] | "Mom" [ "Dad" ] ] .
Child     =   "Helen" | "Margaret" | "Alice" | "Robyn" | "Cathy"
            | "Janet" | "Anne" | "Ntombizodwa" | "Ntombizanele" .
```

Task 6 - Palindromes

Palindromes are character strings that read the same from either end. You were invited to explore various ways of finding grammars that describe palindromes made only of the letters *a* and *b*:

(1) *Palindrome* = "*a*" *Palindrome* "*a*" | "*b*" *Palindrome* "*b*" .
(2) *Palindrome* = "*a*" *Palindrome* "*a*" | "*b*" *Palindrome* "*b*" | "*a*" | "*b*" .
(3) *Palindrome* = "*a*" [*Palindrome*] "*a*" | "*b*" [*Palindrome*] "*b*" .
(4) *Palindrome* = ["*a*" *Palindrome* "*a*" | "*b*" *Palindrome* "*b*" | "*a*" | "*b*"] .

Which grammars achieve their aim? If they do not, explain why not. Which of them are LL(1)? Can you find other (perhaps better) grammars that describe palindromes and which *are* LL(1)?

This is one of those awful problems that looks deceptively simple, and indeed is deceptive. We need to be able to cater for palindromes of odd or even length, and we need to be able to cater for palindromes of finite length, so that the "repetition" that one immediately thinks of has to be able to terminate.

Here are some that don't work:

```
COMPILER Palindrome /* does not terminate */
PRODUCTIONS
  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" .
END Palindrome.
```

```
COMPILER Palindrome /* only allows odd length palindromes */
PRODUCTIONS
  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" | "a" | "b" .
END Palindrome.

COMPILER Palindrome /* only allows even length palindromes */
PRODUCTIONS
  Palindrome = "a" [ Palindrome ] "a" | "b" [ Palindrome ] "b" .
END Palindrome.
```

Of those grammars, the first *seems* to obey the LL(1) rules, but it is useless (it is not "reduced" in the sense of the definitions on page 129). The second one is obviously non-LL(1) as the terminals "a" and "b" can start more than one alternative. The third one is less obviously non-LL(1). If you rewrite it

```
COMPILER Palindrome /* only allows even length palindromes */
PRODUCTIONS
  Palindrome = "a" Extra "a" | "b" Extra "b" .
  Extra      = Palindrome | ε .
END Palindrome.
```

and note that `Extra` is nullable, then `FIRST(Extra) = { "a", "b" }` and `FOLLOW(Extra) = { "a", "b" }`.

Here is another attempt

```
COMPILER Palindrome /* allows any length palindromes */
PRODUCTIONS
  Palindrome = [ "a"  Palindrome  "a" | "b"  Palindrome  "b" | "a" | "b" ] .
END Palindrome.
```

This describes both odd and even length palindromes, but is non-LL(1). `Palindrome` is nullable, and both `FIRST(Palindrome)` and `FOLLOW(Palindrome) = { "a", "b" }`. And, as most were quick to notice, it breaks Rule 1 immediately as well.

Other suggestions were:

```
COMPILER Palindrome /* allows any length palindromes */
PRODUCTIONS
  Palindrome =  "a"  [ Palindrome  "a"] | "b"  [ Palindrome  "b" ] .
END Palindrome.
```

but, ingenious as this appears, it does not work either. Rewritten it would become

```
COMPILER Palindrome /* allows any length palindromes */
PRODUCTIONS
  Palindrome =  "a"  PalA | "b" PalB .
  PalA       = Palindrome  "a" | .
  PalB       = Palindrome  "b" | .
END Palindrome.
```

`PalA` and `PalB` are both nullable, and `FIRST(PalA) = { "a" , "b" }` while `FOLLOW(PalA) = FOLLOW(Palindrome) = { "a", "b" }` as well.

In fact, when you think about it, you simply will not be able to find an LL(1) grammar for this language. (That is fine; grammars don't have to be LL(1) to be valid grammars. They just have to be LL(1) or very close to LL(1) to be able to write recursive descent parsers.) Here's how to think about it. Suppose I asked you to hold your breath for as long as you could, and also to nod your head when you were half way through. I don't believe you could do it - you don't know before you begin exactly how long you will be holding your breath. Similarly, if I told you to get into my car and drive it till the tank was empty but to hoot the hooter when you were half way to running out you could not do it. Or if I told you to walk into a forest with your partner and kiss him/her when you were in the dead centre of the forest, you would not know when the magic moment had arrived.

LL(1) parsers have to be able to decide just by looking at one token exactly what to do next - if they have to *guess* when they are are half-way through parsing some structure they will not be able to do so. One would have to stop applying the options like `Palindrome = "a" Palindrome "a"` at the point where one had generated or analyzed half the palindrome, and if there is no distinctive character in the middle, one would not expect the parser to be able to do so.

If course, if one changes the problem ever so slightly in that way one *can* find an LL(1) grammar. Suppose we want a grammar for palindromes that have matching *a* and *b* characters on either end and a distinctive *c* or pair of *c* characters in the centre:

```
COMPILER Palindrome /* allows any length palindromes, but c must be in the middle */
PRODUCTIONS
  Palindrome = "a"  Palindrome  "a" | "b"  Palindrome  "b" | "c" [ "c" ] .
END Palindrome.
```

Several submissions suggested (but did not justify) that maybe this problem could be solved by using a context-sensitive set of productions (which would not be LL(1)). That may be possible - I must think about it some more. Context-sensitive grammars are awkward to work with!

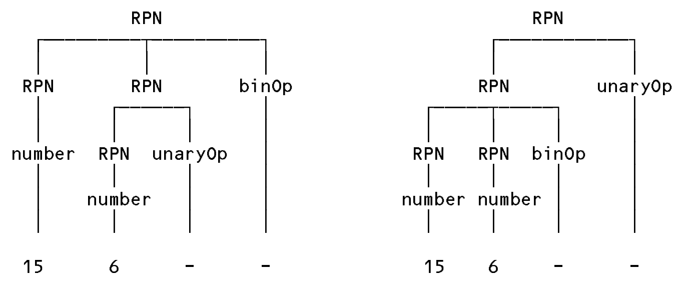
Task 7 - Reverse Polish Notation

The grammars are not equivalent. To show this we need only find one string that one will accept but the other will not, as many people realized. An example of such a string would be `45 sqrt sqrt`, which can only be recognized by G1 in one way, but not at all by G2. Using the expression

```
15  6  -  -
```

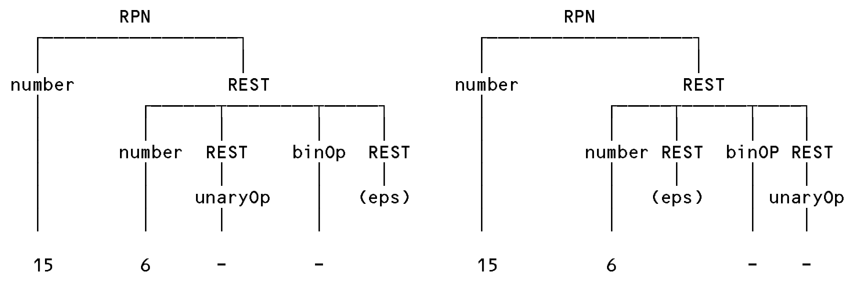
as an example, and by drawing appropriate parse trees, we can demonstrate that both of the grammars are ambiguous.

Using grammar G1:



(implied value is 21)   (implied value is -9)

Using grammar G2:



To analyse each of these grammars to check whether they conform to the LL(1) conditions:

G1 is left recursive and thus is immediately ruled out as a possible LL(1) grammar. There are two alternatives for the right side of the production for RPN that both start with RPN, so Rule 1 is broken, regardless of the exact nature of FIRST(RPN) - which in any case is `number`, and all three alternatives start with this

For G2, REST is nullable. First(REST) is { number, sqrt, - } while Follow(REST) is { +, -, / , * } so Rule 2 is broken.

To try to overcome the problem we seem at the outset to have try to to find a different token to represent the operation of unary negation, as several people realized. If, for example, we use `negate` for the "unary minus", then G1 becomes non-ambiguous (but still not LL(1)) but G2, while now being LL(1), will still not recognize `4 sqrt sqrt`. Like the palindrome example, this one is frustrating - it looks so easy. Again we emphasize that a valid grammar does not *have* to be LL(1) - it just helps when building recursive descent parsers.

Some of students in recent years have suggested better grammars might be as follows. What do you think?

```
PRODUCTIONS
  RPN7    =   number REST .
  REST    = number [ REST ] binOp { unaryOp } | UnaryOp .
  binOp   = "+" | "-" | "*" | "/" .
  unaryOp = "-" | "sqrt" .
END RPN7.

PRODUCTIONS
  RPN8    =   number [ RPN8 binOp ] [ unaryOp ] .
  binOp   = "+" | "-" | "*" | "/" .
  unaryOp = "-" | "sqrt" .
END RPN8.

PRODUCTIONS
  RPN9    =   RPN9 ( unaryOp | RPN9 binOp )  |  number .
  binOp   = "+" | "-" | "*" | "/" .
  unaryOp = "-" | "sqrt" .
END RPN9.
```

Task 8 - Pause for thought

Which of the following statements are true? Justify your answer.

(a) An LL(1) grammar cannot be ambiguous.
(b) A non-LL(1) grammar must be ambiguous.
(c) An ambiguous language cannot be described by an LL(1) grammar.
(d) It is possible to find an LL(1) grammar to describe any non-ambiguous language.

To answer this sort of question you must be able to argue convincingly, and most people did not do that at all!

(a) is TRUE. An LL(1) grammar cannot be ambiguous. If a language can be described by an LL(1) grammar it will always be able to find a single valid parse tree for any valid sentence, and no parse tree for an invalid sentence. The rules imply that no indecision can exist at any stage - either you can find a unique way to continue the implicit derivation from the goal symbol, or you have to conclude that the sentence is malformed.

But you cannot immediately conclude any of the "opposite" statements, other than (c) which is TRUE. If you *really* want to define an ambiguous language (and you may have perfectly good/nefarious reasons for doing so - stand-up comedians do it all the time) you will not be able to describe it by an LL(1) grammar, which has the property that it can only be used for deterministic parsing.

In particular (b) is FALSE. We can "justify" this by giving just a single counter example to the claim that it might be true. We have seen several such grammars. The palindrome grammars above are like this - even though they are non LL(1) for the reasons given, they are quite unambiguous - you would only be able to parse any palindrome in one way! Many people seem not to realize this - they incorrectly conclude that non-LL(1) inevitably implies ambiguity. The other classic case is that of the left-recursive expression grammars discussed in class and in chapter 6.1.

Similarly, the simple grammar

```
Goal = "first" "next" { "next" | "other" } "next" "last" "." .
```

is non-LL(1), but it is not ambiguous - you could only parse the string

```
first  next  next  next  last  .
```

in one way. This is a particularly simple grammar and it is hopefully easy to see that *any* valid string defined by it could only be parsed in one way.

Similarly (d) is FALSE. Once again the palindrome example suffices - this language is simple, unambiguous, but we can easily argue that it is impossible to find an LL(1) grammar to describe it.