

# Computer Science 3 - 2008

## Programming Language Translation

### Practical for Week 22, beginning 22 September 2008 - Solutions

This tutorial/practical was not always well done. Many people could "guess" the answers, but could not or did not justify their conclusions. **If set in exams, in these sorts of questions it is important to do so.**

As usual, you can find a "solution kit" as PRAC22A.ZIP or PRAC22AC.ZIP if you wish to experiment further.

#### Task 1 - 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 `{ }` metabracquets 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 2 - Expressions - again

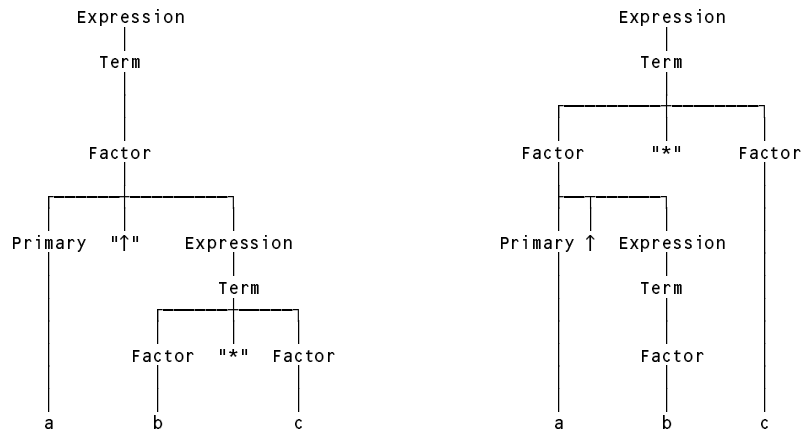
The following grammar attempts to describe expressions incorporating the familiar operators with their correct precedence and associativity.

```
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  $a \uparrow b * c$ . This can indeed be parsed in two ways, one with the implicit meaning of  $a \uparrow (b * c)$  and the other with the meaning of  $(a \uparrow b) * c$ . 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)?

If we rewrite the first grammar to eliminate the metabackets 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 127, 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 3 - Palindromes - again

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  $\text{FIRST}(\text{Extra}) = \{ "a", "b" \}$  and  $\text{FOLLOW}(\text{Extra}) = \{ "a", "b" \}$ .

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  $\text{FIRST}(\text{Palindrome})$  and  $\text{FOLLOW}(\text{Palindrome}) = \{ "a", "b" \}$ . 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  $\text{FIRST}(\text{PalA}) = \{ "a", "b" \}$  while  $\text{FOLLOW}(\text{PalA}) = \text{FOLLOW}(\text{Palindrome}) = \{ "a", "b" \}$  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 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.
```

## Task 4 - 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 seemed not to realise this - they were incorrectly concluding that non-LL(1) inevitably implied ambiguity.)

Similarly, a variation on the train grammar discussed in class and simplified to

```
Train = "loco" "coal" { "coal" | "fuel" } "coal" "guard" "." .
```

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

```
loco coal coal coal guard .
```

in one way. This is a particularly simple grammar and it is hopefully easy to see that *any* valid train 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.

## Task 6 - Deja vu all over again - the joys of CSC 201 assembler

You were invited to develop a grammar for programs written in the CSC 201 toy assembler language, such as exemplified by

```

        BEG          ; Count the bits in a number
        CLA          ; CPU.A := 0
        STA     BITS ; BITS := 0
        INI          ; Read(CPU.A)
LOOP    ; REPEAT
        SHR          ; CPU.A := CPU.A DIV 2
        BCC     EVEN ; IF CPU.A MOD 2 # 0 THEN
        PSH          ; save CPU.A on stack
        LDA     BITS
        INC
        STA     BITS ; BITS := BITS + 1
        POP          ; restore CPU.A
EVEN    BNZ     LOOP ; UNTIL CPU.A = 0
        LDA     BITS
        OTI          ; Write(BITS)
        HLT          ; terminate execution
BITS    DS      1    ; BYTE BITS
END

```

There were a whole lot of points missed, understandably, since (a) you had forgotten and/or (b) might never have explored the vagaries of this language fully.

One important feature, which most groups realized, was that one should split the opcodes into two groups - those that require an Address field, and those that do not. Relatively few understood how the address field could be defined. As is typical of many assembler systems, the expressions here can mix numbers, labels, strings and the special term denoted by \* (few people had got close to understanding this last one).

Finally, in assembler languages the rule is almost invariably at most "one statement per line". But lines can be devoid of opcodes, and have only labels or comments on them. So we need to be careful to capture this idea too.

Here is a possible first attempt at a grammar.

```

COMPILER ASM1 $NC

CHARACTERS
  lf      = CHR(10) .
  cr      = CHR(13) .
  control = CHR(0) .. CHR(31) .
  letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit   = "0123456789" .
  binDigit = "01" .
  hexDigit = digit + "ABCDEFabcdef" .
  charCh  = ANY - " " - control .
  printable = ANY - control .

TOKENS
  number  = digit { digit } | binDigit { binDigit } "%" | digit { hexDigit } "H" .
  identifier = letter { letter | digit } .
  string  = '"' charCh { charCh } '"' .
  EOL     = cr lf | lf .
  comment = ";" { printable } .

IGNORE control - cr - lf

PRODUCTIONS
  ASM1      = StatementSequence .
  StatementSequence = { Statement [ comment ] SYNC EOL } .
  Statement = [ Label ] [ OneByteOp | TwoByteOp Address ] .
  OneByteOp =
    "ASR" | "BEG" | "CLA" | "CLC" | "CLV" | "CLX" | "CMA"
    "CMC" | "DEC" | "DEX" | "END" | "HLT" | "INA" | "INB"
    "INC" | "INH" | "INI" | "INX" | "NOP" | "OTA" | "OTB"
    "OTC" | "OTH" | "OTI" | "POP" | "PSH" | "RET" | "SHL"
    "SHR" | "TAX" .
  TwoByteOp =
    "ACI" | "ACX" | "ADC" | "ADD" | "ADI" | "ADX" | "ANA"
    "ANI" | "ANX" | "BCC" | "BCS" | "BGE" | "BGT" | "BLE"
    "BLT" | "BNG" | "BNZ" | "BPZ" | "BRN" | "BVC" | "BVS"
    "BZE" | "CMP" | "CPI" | "CPX" | "DC" | "DS" | "EQU"
    "JGE" | "JGT" | "JLE" | "JLT" | "JSR" | "LDA" | "LDI"
    "LDX" | "LSI" | "LSP" | "ORA" | "ORG" | "ORI" | "ORX"
    "SBC" | "SBI" | "SBX" | "SCI" | "SCX" | "STA" | "STX"
    "SUB" .
  Address = Term { '+' Term | '-' Term } .
  Term    = Label | number | string | '*' .
  Label   = identifier .
END ASM1.

```

This makes use of the Coco/R directive SYNC that we have not discussed yet - it signals a convenient synchronization point. If no EOL is found where one should occur, the scanner simply consumes tokens until the EOL is found - this is a simple and very effective error recovery technique for "one statement per line" systems.

As an alternative, one might have defined

```
COMMENTS FROM ";" TO Lf
PRODUCTIONS
...
StatementSequence = { Statement SYNC EOL } .
```

The grammar is actually rather too accommodating, as some people realised. Strictly we should not allow `BEG` and `END` to appear more than once each, in very definite places. Directives like `EQU` must have labels, while `ORG` must not have a label - a statement sequence like

```
GO   ORG   100
MAX  EQU   12
      EQU   13
MIN  EQU   14
```

must be meaningless. And strings of length greater than 1 are really only meaningful in `DC` directives.

Enforcing all these constraints is tricky, and in the case of the `DC` limitation I think impossible if one wants to keep an LL(1) grammar. (There is another way around that problem that we can explore in a future practical). The `BEG/END` restriction is possible syntactically, though its a bit messy, since we must allow for leading blank lines before `BEG` and for trailing blank lines after `END`. Careful refactoring of the grammar sorts out the `EQU` and `ORG` complications. If we retain "comment" as a token we might get

```
PRODUCTIONS
ASM4      = Begin StatementSequence End.
Begin     = { [ comment ] EOL } "BEG" [ comment ] EOL .
End       = "END" [ comment ] EOL { [ comment ] EOL } .
StatementSequence = { [ Statement ] [ comment ] EOL } .
Statement = Label ( "EQU" KnownAddress | OneByteOp | TwoByteOp Address | )
OneByteOp =
  "ASR" | "CLA" | "CLC" | "CLV" | "CLX" | "CMA" | "CMC" |
  "DEC" | "DEX" | "HLT" | "INA" | "INB" | "INC" | "INH" |
  "INI" | "INX" | "NOP" | "OTA" | "OTB" | "OTC" | "OTH" |
  "OTI" | "POP" | "PSH" | "RET" | "SHL" | "SHR" | "TAX" .
TwoByteOp =
  "ACI" | "ACX" | "ADC" | "ADD" | "ADI" | "ADX" | "ANA" |
  "ANI" | "ANX" | "BCC" | "BCS" | "BGE" | "BGT" | "BLE" |
  "BLT" | "BNG" | "BNZ" | "BPZ" | "BRN" | "BVC" | "BVS" |
  "BZE" | "CMP" | "CPI" | "CPX" | "DC" | "DS" | "JGE" |
  "JGT" | "JLE" | "JLT" | "JSR" | "LDA" | "LDI" | "LDX" |
  "LSI" | "LSP" | "ORA" | "ORI" | "ORX" | "SBC" | "SBI" |
  "SBX" | "SCI" | "SCX" | "STA" | "STX" | "SUB" .
Address   = Term { '+' Term | '-' Term } .
Term      = Label | number | string | '*' .
Label     = identifier .
KnownAddress = Address .
END ASM4.
```

or, with a bit more factoring out of common components, we could write the first of these as

```
ASM5      = Begin StatementSequence End.
Begin     = { CommentEOL } "BEG" CommentEOL .
End       = "END" CommentEOL { CommentEOL } .
StatementSequence = { [ Statement ] CommentEOL } .
CommentEOL = [ comment ] EOL .
```

It would also make sense to introduce the directive `IGNORECASE` at the start of the grammar, before the `CHARACTERS` section, as assembler languages are usually case insensitive (another Coco/R feature we have not encountered before).

At first I thought one would not be able to use the `COMMENT` directive in Coco/R to handle this sort of language. Surely if one defines



then the scanner should have consumed the LF as part of the comment, and so looking for another one as an EOL token immediately afterwards suggests that one would get into trouble. As it happens (and as several people discovered) it does work - to try to handle the different ways in which operating systems record line breaks (CR on an Apple, LF on Unix and CR/LF on Windoze) Coco/R has a special way of handling the LF behind your back.

The grammar above has used a token definition for `number`

```
TOKENS
  number = digit { digit } | binDigit { binDigit } "%" | digit { hexDigit } "H" .
```

This is quite adequate here, but it would be preferable to define three different tokens if one was interested in determining the underlying values - as one would be in a real assembler that generated code.

```
TOKENS
  decNumber = digit { digit } .
  binNumber = binDigit { binDigit } "%" .
  hexNumber = digit { hexDigit } "H" .
```

## Task 7 - Parva expressions are not like those in C# and Java

The grammar for expressions in Parva employs very few levels of operator precedence, corresponding exactly to the levels found in Pascal and its derivatives. You were asked to modify the Parva grammar from last week's practical so that it recognizes expressions whose precedence rules are equivalent to those found in C# or Java.

I think most people got at least part of the way there. The correct answer would be as below.

```
COMPILER Parva $CN
.....
Condition      = Expression .
Expression     = AndExp { "||" AndExp } .
AndExp         = EqlExp { "&&" EqlExp } .
EqlExp         = RelExp { EqlOp RelExp } .
RelExp         = AddExp [ RelOp AddExp | "in" ExpList ] .
AddExp         = MulExp { AddOp MulExp } .
MulExp         = Factor { MulOp Factor } .
Factor         = Primary
                | "+" Factor
                | "-" Factor
                | "!" Factor .
Primary        = Designator | Constant
                | "new" BasicType "[" Expression "]"
                | "(" Expression ")" .
AddOp          = "+" | "-" .
MulOp          = "*" | "/" | "%" .
EqlOp          = "==" | "!=" .
RelOp          = ">" | ">=" | "<" | "<=" .
END Parva.
```

A few points were missed by many people. Note that the rule for `RelExp` allows only one further component - this prevents expressions from being formed like

`a < b < c > d`

which could have no meaning. Very few, if any, submissions seemed to have picked up this point.

We divide the "equality" operators from the "relational" operators because expressions like

`a == b == true != false`

could have meaning - and we conveniently include the `in` operator (which is not found in C of course) as another relational operator. Finally, note the way in which the unary operators `+`, `-` and `!` enter the precedence

hierarchy. This grammar allows expressions to be formed like

`a + - + b`

(whatever turns you on!) which is forbidden in Pascal.

You were asked to consider why languages derived from C have so many levels of precedence and the rules they have for associativity, and what the advantages and disadvantages might be over languages like Pascal that get away with fewer.

I guess the designers of C thought that this would be a good idea because it allows you to write statements like

```
if ( a > b and c < d ) DoSomething();
```

which in a Pascal-oriented system would have to be expressed with more parentheses:

```
if ( ( a > b ) and ( c < d ) ) DoSomething();
```

(Of course, one sensibly inserts extra parentheses into any expression if one is not sure of the precedence rules!)

There is another point that one can make, however. Now that you have seen how recursive descent parsers work, you should be able to appreciate that if a C-like compiler for Parva has to recognize the simple expression "a" it must go through a sequence of eight function calls - first to `Expression()`, then to `AndExp()`, then to `OrExp()`, then to `EqExp()` ... until it gets to `Primary()`. In the original grammar it would have to make only four function calls. Function calls take space, and they take time. In the 70's, when memory was expensive and tight and computers were slow, it was a distinct advantage to have small, tight parsers. Niklaus Wirth is a role model for folk like myself of the "keep it as simple as you can" persuasion - I am sure that he would have made a very definite engineering decision.