

Computer Science 3 - 2017

Programming Language Translation

Practical for Week 3, beginning 14 September 2017 - Solutions

Complete sources to these solutions can be found on the course WWW pages in the files PRAC3A.ZIP.

Task 2 - Extensions to the Simple Calculator

In the source kit you were given Calc.atg and you were invited to extend it to allow for parentheses, other forms of hex numbers, factorials and an absolute function.

```
COMPILER calc $CN
/* Simple four function calculator
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  digit    = "0123456789" .
  hexdigit = digit + "ABCDEF" .

TOKENS
  decNumber = digit { digit } .
  hexNumber = "$" hexdigit { hexdigit } | "0" { hexdigit } "H" .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Calc    = { Expression "=" } EOF .
  Expression = Term { "+" Term | "-" Term } .
  Term     = Factor { "*" Factor | "/" Factor } .
  Factor   = Primary { ! } .
  Primary  = decNumber | hexNumber
            | "(" Expression ")"
            | "abs" "(" Expression ")".

END calc.
```

Task 3 - Meet the family

This was meant to be relatively straightforward and should not have caused too many difficulties. A criticism of several submissions was that they were too restrictive. Here is one solution in the spirit of the exercise:

```
COMPILER Family1 $CN
/* Describe a family
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
  control    = CHR(0) .. CHR(31) .
  uLetter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" .
  lLetter    = "abcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .

TOKENS
  name       = uLetter { lLetter | "'" uLetter | "-" uLetter } .
  number     = digit { digit } .

IGNORE control

PRODUCTIONS
  Family1    = { Section SYNC } EOF .
  Section    = Surname | Parents | Grandparents | Children | Grandchildren | Possession .
  Surname    = "Surname" ":" name { name } .
  Parents    = "Parents" ":" PersonList .
  Grandparents = "Grandparents" ":" PersonList .
  Children   = "Children" ":" PersonList .
  Grandchildren = "Grandchildren" ":" PersonList .
  PersonList = OnePerson { "," OnePerson } .
  OnePerson  = name { name } [ "(" "deceased" ")" ] { Description } [ Spouse ] .
  Spouse     = "=" name { name } .
  Description = "[" Relative "of" OnePerson "]" .
  Relative   = "son" | "daughter" | "mother" | "father"
              | "wife" | "husband" | "partner" | "mistress" .
  Possession = number [ "small" | "large" ]
                ( "cat" | "cats" | "dog" | "dogs" | "bagpipe" | "bagpipes"
                  | "house" | "houses" | "car" | "cars" ) .

END Family1.
```

That solution does not insist that the surname should be part of all descriptions. Here is an alternative PRODUCTIONS set that does just that, and also factorizes the grammar slightly differently:

```
PRODUCTIONS
Family2      = { Generation SYNC } Surname SYNC { Generation SYNC } { Possession } EOF .
Surname      = "Surname" ":" name { name } .
Generation   = ( "Parents" | "Grandparents" | "Children" | "Grandchildren" ) ":" PersonList .
PersonList   = OnePerson { "," OnePerson } .
OnePerson    = name { name } [ "(" "deceased" ")" ] { Description } [ Spouse ] .
Spouse       = "=" name { name } .
Description  = "[" Relative "of" OnePerson "]" .
Relative     = "son" | "daughter" | "mother" | "father"
              | "wife" | "husband" | "partner" | "mistress" .
Possession   = number [ "small" | "large" ]
              ( "cat" | "cats" | "dog" | "dogs" | "bagpipe" | "bagpipes"
                | "house" | "houses" | "car" | "cars" ) .

END Family2.
```

Four points are worth making (a) the Surname section should not have allowed the possibility of listing the name as deceased (b) it is better to use a construct like "(" "deceased" ")" than "(deceased)" as a single terminal (c) relationships are best between OnePerson and another OnePerson, and not simply between OnePerson and some names (d) there is no need to make line feeds significant in this example - although no harm is done if you do, and they certainly make the text easier for a human reader to decode.

Note how we have defined "cat" and "cats" as keywords. We might alternatively have introduced a token

```
item = LLetter { LLetter } .
```

and changed the production to allow for all sorts of other goodies!

```
Possession = number [ "small" | "large" ] item .
```

Task 4 - So what if Parva is so restrictive - fix it!

The Parva extensions produced some interesting submissions. Many of them (understandably!) were too restrictive in certain respects, while others were too permissive. Admittedly there is a thin line between what might be "nice to have" and what might be "sensible to have" or "easy to compile". Here is a heavily commented suggested solution.

```
COMPILER Parva $CN
/* Parva level 1.5 grammar (Extended)
   This version uses Pascal/Modula-like precedences for operators
   P.D. Terry, Rhodes University, 2017 */

CHARACTERS
lf      = CHR(10) .
backslash = CHR(92) .
control = CHR(0) .. CHR(31) .
letter  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
digit   = "0123456789" .
nonZeroDigit = "123456789" .
binDigit = "01" .
hexDigit = digit + "abcdefABCDEF" .
stringCh = ANY - "'" - control - backslash .
charCh   = ANY - '"' - control - backslash .
printable = ANY - control .

TOKENS

identifier = letter { letter | digit | "_" } .

number     = "0" | nonZeroDigit { digit } .

/* But be careful. There is a temptation to define
   digit = "123456789" .
   number = "0" | digit { digit | "0" } .
   and then forget that
   identifier = letter { letter | digit | "_" } .
   would not allow identifiers to have 0 in them */
```

```

    stringLit = ''' { stringCh | backslash printable } ''' .
    charLit   = '"' { charch   | backslash printable } '"' .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE control

PRODUCTIONS
    Parva          = "void" identifier "(" ")" Block .
    Block          = "{" { Statement } "}" .

/* The options in Statement are easily extended to handle the new forms */

    Statement      = (
        Block
        | AssignmentStatement
        | ConstDeclarations
        | IfStatement
        | ReturnStatement
        | ReadStatement
        | ReadLineStatement
        | ForStatement
        | ContinueStatement
        | InclStatement
        | VarDeclarations
        | WhileStatement
        | HaltStatement
        | WriteStatement
        | WriteLineStatement
        | BreakStatement
        | RepeatStatement
        | ExclStatement
        | ";"
    ) .

/* Declarations remain the same as before */

    ConstDeclarations = "const" OneConst { "," OneConst } ";" .
    OneConst          = identifier "=" Constant .
    Constant          = number | charLit | "true" | "false" | "null" .
    VarDeclarations   = Type OneVar { "," OneVar } ";" .
    OneVar            = identifier [ "=" Expression ] .

/* AssignmentStatements require care to avoid LL(1) problems */

    AssignmentStatement = (
        Designator ( AssignOp Expression | "++" | "--" )
        | "++" Designator
        | "--" Designator
    ) ";" .

/* In all these it is useful to maintain generality by using Designator, not identifier */

    Designator        = identifier [ "[" Expression "]" ] .

/* The if-then-else construction is most easily described as follows. Although
   this is not LL(1), this works admirably - it is simply the well-known dangling
   else ambiguity, which the parser resolves by associating the else clauses
   with the most recent if */

    IfStatement       = "if" "(" Condition ")" Statement
                       [ "else" Statement ] .

/* Remember that the RepeatStatement must end with a semicolon - easy to forget this! */

    RepeatStatement    = "repeat" { Statement } "until" "(" Condition ")" ";" .

/* Break and continue statements are very simple. They are really "context dependent" but we
   cannot impose such restrictions in a context free grammar */

    BreakStatement     = "break" ";" .
    ContinueStatement  = "continue" ";" .

/* ReadLine and WriteLine statements must allow for an empty argument list */

    ReadLineStatement  = "readLine" "(" [ ReadElement { "," ReadElement } ] ")" ";" .
    WriteLineStatement = "writeLine" "(" [ WriteElement { "," WriteElement } ] ")" ";" .

/* The incl and excl statements are very easy */

    InclStatement      = "incl" "(" Designator "," Expression ")" ";" .
    ExclStatement      = "excl" "(" Designator "," Expression ")" ";" .

/* Alternatively one might generalise these statements for wider use

    InclStatement      = "incl" "(" Designator "," ExprList ")" ";" .
    ExclStatement      = "excl" "(" Designator "," ExprList ")" ";" .

Note that here we would not be wise to try to generalise further to

```

```

InclStatement      = "incl" "(" Expression "," ExprList ")" ";" .
ExclStatement      = "excl" "(" Expression "," ExprList ")" ";" .

Although a Designator can be seen as a special case of an Expression, statements like

        incl(Set1 + Set2, 7);
        incl(24, 7),

would, in the first case, add the 7 to a dynamic "union" for which there is no
handle/pointer/variable, and in the second case could have no meaning at all. */

/* The suggested form of the ForStatement could be described by the following */

ForStatement       = "for" identifier "in" "(" ExprList ")" Statement .

/* Much of the rest of the grammar remains unchanged: */

WhileStatement     = "while" "(" Condition ")" Statement .
ReturnStatement    = "return" ";" .
HaltStatement      = "halt" ";" .
ReadStatement      = "read" "(" ReadElement { "," ReadElement } ")" ";" .
ReadElement        = stringLit | Designator .
WriteStatement     = "write" "(" WriteElement { "," WriteElement } ")" ";" .
WriteElement       = stringLit | Expression .
Condition          = Expression .

Expression         = AddExp [ RelOp AddExp ] .
AddExp             = [ "+" | "-" ] Term { AddOp Term } .
Term               = Factor { MulOp Factor } .
Factor             =
    Designator
    | Constant
    | SetFactor
    | "new" BasicType "[" Expression "]"
    | "!" Factor | "(" Expression ")" .

/* The ExprList used to define a set factor can be quite powerful, syntactically */

SetFactor          = "(" [ ExprList ] ")" .
ExprList           = Range { "," Range } .
Range              = Expression [ ".." Expression ] .

/* It might be thought that a suitable alternative to this might be

ExprList           = Expression { ( ".." | "," ) Expression } .

and, indeed, syntactically that production is equivalent to the pair for ExprList and
Range. But, just as multiplication and division take precedence over addition and
subtraction, so the computation of a range will give an implicit list of Expressions
which will all have to be evaluated before they are included in the overall SetFactor */

Type               = BasicType [ "[" ] ] .

/* To the basic types we must add set and char */

BasicType          = "int" | "bool" | "char" | "set" .

/* We add the pesky % to the mulops */

AddOp              = "+" | "-" | "||" .
MulOp              = "*" | "/" | "%" | "&&" .

/* The "in" operator is just another relational operator (many don't realise this) */

RelOp              = "==" | "!=" | "<" | "<=" | ">" | ">=" | "in" .
AssignOp           = "=" .

END Parva .

```

Task 5 - Transnet are looking for programmers

You were asked to develop a grammar to describe various kinds of trains, subject to the restrictions that fuel trucks may not be marshalled immediately behind the locomotives, or immediately in front of a passenger coach.

In my experience the wheels come off in many attempts at solving this problem. It is quite hard to get right, and at first one may not easily find an LL(1) grammar that really matches the problem as set.

Given that passenger trains do not have a safety complication, one might be tempted to refactor the grammar to

give the equivalent one below, which seems more closely to define a train in terms of the three ways in which it can be classified. Note that this does not yet attempt to impose the safety conditions.

```

COMPILER Train1 $CN
/* Grammar for simple railway trains
   P.D. Terry, Rhodes University, 2017 */

IGNORECASE

COMMENTS FROM "(" TO ")" NESTED

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Train1      = { OneTrain } EOF .
  OneTrain    = LocoPart [ Passengers | FreightOrMixed ] SYNC "." .
  LocoPart    = "loco" { "loco" } .
  FreightOrMixed = Truck { Truck } ( "brake" | Passengers ) .
  Passengers  = { "coach" } "guard" .
  Truck       = "closed" | "coal" | "open" | "cattle" | "fuel" .
END Train1.

```

Here is an attempt at safety. But this one insists on at least two safe trucks in any train, and is not LL(1):

```

PRODUCTIONS
  Train2      = { OneTrain } EOF .
  OneTrain    = LocoPart [ Passengers | FreightOrMixed ] SYNC "." .
  LocoPart    = "loco" { "loco" } .
  FreightOrMixed = SafeTruck { AnyTruck } LastPart .
  LastPart    = "brake" | SafeTruck Passengers .
  Passengers  = { "coach" } "guard" .
  SafeTruck   = "closed" | "coal" | "open" | "cattle" .
  AnyTruck    = SafeTruck | "fuel" .
END Train2.

```

Why is it not LL(1) compliant? We could apply all the theory of Chapter 7 of the textbook, but maybe an example will suffice as an alternative. Suppose we have a valid train like

```
loco coal coal coal coal coach guard
```

The first coal truck is parsed by the leading SafeTruck in GoodsPart. The next two coal trucks must be parsed by the repetitive part { AnyTruck }, but you can probably see that the last coal truck would have to be parsed by the alternative within LastPart. Unfortunately an LL(1) parser can't see far enough ahead to make that decision, and would be tempted to treat this last coal truck as part of the { AnyTruck } sequence.

Using the ideas developed in section 7.3 of the text is straightforward. Rule 1 is clearly satisfied for the productions for OneTrain, LastPart, SafeTruck and AnyTruck. Rule 2 - the FIRST/FOLLOW rule as applied to nullable sections - is easily seen to be satisfied for the nullable sections in the productions for Train2, OneTrain and Passengers. However, for the nullable section in the production for FreightOrMixed, FIRST({Anytruck}) and FIRST(LastPart) both include the set FIRST(SafeTruck), that is { coal, closed, open, cattle }.

Here is one that *is* LL(1)

```

PRODUCTIONS
  Train3      = { OneTrain } EOF .
  OneTrain    = LocoPart [ Passengers | FreightOrMixed ] SYNC "." .
  LocoPart    = "loco" { "loco" } .
  FreightOrMixed = SafeTruck MoreTrucks HumanPart .
  MoreTrucks  = { "fuel" { "fuel" } SafeTruck | SafeTruck } .
  HumanPart   = "brake" | Passengers .
  Passengers  = { "coach" } "guard" .
  SafeTruck   = "coal" | "closed" | "open" | "cattle" .
END Train3.

```

At first you might think that this is, at last, a correct solution. But no, it isn't quite. This solution does not allow you to have a train like:

```
loco loco open fuel fuel brake .
```

as the last fuel truck in a sequence has now to be followed by at least one safe truck. The grammar does, however, allow trains like

```
Loco open coach coach guard .
```

with only one truck in the freight section.

It is remarkable that something that at first sight looks as though it would be simple might turn out to be frustratingly difficult. Not being able to find an LL(1) grammar is not a train smash - one quite often cannot find an LL(1) grammar for a language. But it's usually worth a try, as parsers for LL(1) grammars are so easy to write. The clue is to be found in a suggestion that one should factorize the grammar not to concentrate on the "obvious" kinds of train, but on the requirement that at any point along a train that might incorporate fuel trucks, the last part of the train should be "safe". Thus:

```
PRODUCTIONS
Train4      = { OneTrain } EOF .
OneTrain    = LocoPart [ SafeLoad | "brake" | Passengers ] SYNC "." .
LocoPart    = "loco" { "loco" } .
Passengers  = { "coach" } "guard" .
SafeLoad    = SafeTruck { SafeTruck } ( "brake" | Passengers | SafeFuel ) .
SafeFuel    = "fuel" { "fuel" } ( SafeLoad | "brake" ) .
SafeTruck   = "coal" | "closed" | "open" | "cattle" .
END Train4.
```

Each year, when I give the compiler course, some students come up with better solutions to problems than I had not thought of, and one year I was intrigued by the following solution to this problem. I am not sure which of the five students who seem to have contributed to the solution below had the "Aha!" moment, but I was very impressed and hope that it really was a joint effort. This solution makes neat use of right recursion without using the {..} meta-brackets, a technique that is worth bearing in mind.

```
COMPILER Train5 $CN
/* Grammar for safe railway trains
   Original by P.D. Terry, Rhodes University,
   This idea based on one by Mikha Zeffert, Jessica Kent, Alisa Lochner,
   Kelvin Freese and Michael Abbott, 2013 class */

IGNORECASE

COMMENTS FROM "(" TO ")" NESTED

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
Train5      = { OneTrain } EOF .
OneTrain    = LocoPart [ GoodsPart | HumanPart | "brake" ] SYNC "." .
LocoPart    = "loco" { "loco" } .
HumanPart   = { "coach" } "guard" .
GoodsPart   = SafeTruck [ HumanPart | GoodsPart | FuelPart | "brake" ] .
FuelPart    = "fuel" [ GoodsPart | FuelPart | "brake" ] .
SafeTruck   = "coal" | "closed" | "open" | "cattle" .
END Train5.
```

Terry Theorem One ("You can always tidy things up a bit more") might apply, and here is a slightly refactored version of the productions.

```
PRODUCTIONS
Train6      = { OneTrain } EOF .
OneTrain    = LocoPart [ GoodsPart | HumanPart | "brake" ] SYNC "." .
LocoPart    = "loco" { "loco" } .
HumanPart   = { "coach" } "guard" .
GoodsPart   = SafeTruck [ HumanPart | SafePart ] .
SafePart    = GoodsPart | FuelPart | "brake" .
FuelPart    = "fuel" [ SafePart ] .
SafeTruck   = "coal" | "closed" | "open" | "cattle" .
END Train6.
```

The 2017 class were very active in trying to solve this problem too. The solution kit for this prac PRAC3A.ZIP on the website includes several grammars submitted by students in the 2017 class. You might like to study them and decide which, if any, are correct, and which are incorrect, and why!