Adapting Grammar to a LARL(1) Parser

vpcepa@yahoo.com

Our example grammar generates strings that look like this:

```
int a = 2, b;
int c;
real d = 2.3, e = 5, f;

int f1(){
...
}

real f2(int a, real b){
...
}
```

The first solution that comes to mind but that is not LARL(1) parsable, is shown below in the yacc notation:

```
vardecls: vardecls ',' vardec
        | vardec
        | /* empty */
        ;

vardecl: vartype ID
        | vartype ID '=' expr
        ;

vartype: INT
        | REAL
        ;

expr: CINT
        | CREAL
        ;

funcdecls: funcdecls funcdec
        | funcdec
        | /* empty */
        ;

funcdec: vartype ID '(' arglist ')' block;
        ;

arglist: arglist arg
        | arg
        | /* empty */
        ;

arg: vartype ID
        ;

block: ...
```

```
INT: int
REAL: real
CINT: any int constant
CREAL: any real constant
```

The problem is that lookahead token for *funcdec* and *vardec* is equal. To make this grammar yacc parsable we have to find what makes the token different, if we see only one token ahead at a time. To do this, we consider the following list of possibilities generate by the above grammar:

```
vartype ID ';'
vartype ID ',' ID ';'
vartype ID '=' expr ',' ID = expr ';'
vartype ID '(' ...
```

What distinguishes the sentences with only one look ahead is the token next to ID. This is the key token we must group with the ID. In particular we have a list of variable declarations that has a start, middle and an end. For the example above we have:

```
startlist: ID ','
        | ID '='
        ;
```

We will call the second production *varidexpr* so we have these grammar slices:

```
startlist: ID ','
        | ID varidexpr
        ;

endlist: ID ';'
        | ID varidexpr ';'
        ;
```

If we look more carefully at our examples we can note that middle production of the list is actually repetition of the start list. Finally our modified *vardec* rule becomes:

```
vardec -> vartype varidlist
        ;

varidlist -> startlistlist endvarid
        | endvarid
        ;

startlistlist -> startlistlist startlist
        | startlist
        ;

startlist: ID ','
        | ID varidexpr
        ;

endlist: ID ';'
        | ID varidexpr ';'
        ;

varidexpr -> '=' expr
        ;
```

Then we create a top declaration rule that groups the possibilities. This factorizes the need for multiple empty clauses that were the source of conflicts:

```
decls -> vardecls funcpdecs funcdecls
        | vardecls funcdecls
        | vardecls
        | funcdecls
        | /* empty */
        ;
```

We use then the group ID '(' to distinguish functions in the same fashion as above. We denote it as *fid* in a separate rule here; however this is not strictly necessary. Thus the complete new function rules become as follows:

```
funcdecls -> funcdecls funcdec
        | funcdec
        ;

funcdec -> funchead arglist ')' block
        ;

funchead -> vartype fid
        ;

fid -> ID '('
        ;
```

The *arglist* remains the same. Our new grammar contains now only shift/reduce rules that default to what a yacc compatible parser expects.

EOF