Computer Science 3 - Programming Language Translation

Adding the character type to Parva

To allow for a character type in Parva involves several straightforward alterations, as well as some more elusive ones. Firstly, we extend the definition of a symbol table entry type (code samples here are expressed in C#, but are effectively the same in Java):

```
        class Types {
          public const int
            noType   =  0,             // identifier (and expression) types.  The numbering is
            nullType =  2,             // significant as array types are denoted by these
            intType  =  4,             // numbers + 1
            boolType =  6,
   *        charType =  8,
   *        voidType = 10;
          ...
        } // end Types
```

The *Table* class requires a similar small change to introduce the new type name needed if the symbol table is to be displayed:

```
   *  typeNames.Add("char");
   *  typeNames.Add("char[]");
```

A minor change to the *Constant* production is needed to allow character literals to be regarded as of the new `charType`:

```
        Constant<out ConstRec con>           (. con = new ConstRec(); .)
        =   IntConst<out con.value>          (. con.type = Types.intType; .)
   *      | CharConst<out con.value>         (. con.type = Types.charType; .)
          | "true"                           (. con.type = Types.boolType; con.value = 1; .)
          | "false"                          (. con.type = Types.boolType; con.value = 0; .)
          | "null"                           (. con.type = Types.nullType; con.value = 0; .) .
```

Dealing with reading and writing single characters is easy:

```
        ReadElement                          (. string str;
                                                DesType des; .)
   *    =   StringConst<out str>             (. CodeGen.WriteString(str); .)
          | Designator<out des>              (. if (des.entry.kind != Kinds.Var)
                                                  SemError("wrong kind of identifier");
   *                                            if (!des.canChange)
   *                                              SemError("may not alter this variable");
                                                switch (des.type) {
                                                  case Types.intType:
                                                  case Types.boolType:
   *                                              case Types.charType:
                                                    CodeGen.Read(des.type); break;
                                                  default:
                                                    SemError("cannot read this type"); break;
                                                } .) .


        WriteElement                         (. int expType;
                                                string str; .)
   *    =   StringConst<out str>             (. CodeGen.WriteString(str); .)
          | Expression<out expType>          (. switch (expType) {
                                                  case Types.intType:
                                                  case Types.boolType:
   *                                              case Types.charType:
                                                    CodeGen.Write(expType); break;
                                                  default:
                                                    SemError("cannot write this type"); break;
                                                } .) .
```

The associated code generating methods require matching additions:

```
       public static void Read(int type) {
       // Generates code to read a value of specified type
       // and store it at the address found on top of stack
         switch (type) {
           case Types.intType:  Emit(PVM.inpi); break;
           case Types.boolType: Emit(PVM.inpb); break;
   *       case Types.charType: Emit(PVM.inpc); break;
         }
       }

       public static void write(int type) {
       // Generates code to output value of specified type from top of stack
         switch (type) {
           case Types.intType:  Emit(PVM.prni); break;
           case Types.boolType: Emit(PVM.prnb); break;
   *       case Types.charType: Emit(PVM.prnc); break;
         }
       }
```

The major part of this exercise is concerned with the changes needed to apply various constraints on operands of the `char` type. Essentially, it ranks as an arithmetic type, in that expressions of the form

character + character
character > character
character + integer
character > integer

are all allowable. This can be handled by modifying the helper methods in the parser as follows:

```
       static bool IsArith(int type) {
   *     return type == Types.intType || type == Types.charType || type == Types.noType;
       }

   *   static bool Compatible(int typeOne, int typeTwo) {
   *   // Returns true if typeOne is compatible (and comparable for equality) with typeTwo
         return    typeOne == typeTwo
   *            || IsArith(typeOne) && IsArith(typeTwo)
                || typeOne == Types.noType || typeTwo == Types.noType
                || IsRef(typeOne) && typeTwo == Types.nullType
                || IsRef(typeTwo) && typeOne == Types.nullType;
       }
```

However, assignment compatibility is more restrictive. Assignments of the form

integer = integer expression
integer = character expression
character = character expression

are allowed, but

character = integer expression

is not allowed. This may be checked within the *Assignment* production with the aid of a further helper method `assignable`:

```
   *   static bool Assignable(int typeOne, int typeTwo) {
   *   // Returns true if typeOne may be assigned a value of typeTwo
   *     return    typeOne == typeTwo
   *            || typeOne == Types.intType && typeTwo == Types.charType
   *            || typeOne == Types.noType || typeTwo == Types.noType
   *            || IsRef(typeOne) && typeTwo == Types.nullType;
   *   }
```

The `assignable()` function call now takes the place of the `compatible()` function call in the places in *OneVar* and *Assignment* where, previously, calls to `compatible()` appeared.

We turn finally to consideration of the changes needed to the various sub-parsers for expressions.

A casting mechanism is introduced to handle the situations where it is necessary explicitly to convert integer values to characters, so that

character = (char) integer

is allowed, and for completeness, so are

integer   = (int) character
integer   = (char) character
character = (char) character

Casting operations are accompanied by a type check and a type conversion; the `(char)` cast also introduces the generation of run-time code for checking that the integer value to be converted lies within range.

This is all handled within the *Primary* production, which has to be re-factored to deal with the potential LL(1) trap in distinguishing between components of the form `"(" "char" ")"` and `"(" Expression ")"`:

```
    Primary<out int type>             (. type = Types.noType;
                                         int size;
                                         DesType des;
                                         ConstRec con; .)
    =    Designator<out des>          (. type = des.type;
                                         switch (des.entry.kind) {
                                           case Kinds.Var:
                                             CodeGen.Dereference();
                                             break;
                                           case Kinds.Con:
                                             CodeGen.LoadConstant(des.entry.value);
                                             break;
                                           default:
                                             SemError("wrong kind of identifier");
                                             break;
                                         } .)
       | Constant<out con>            (. type = con.type;
                                         CodeGen.LoadConstant(con.value); .)
       | "new" BasicType<out type>    (. type++; .)
         "[" Expression<out size>     (. if (!IsArith(size))
                                           SemError("array size must be integer");
                                         CodeGen.Allocate(); .)
         "]"
*      | "("
*        (   "char" ")"
*            Factor<out type>         (. if (!IsArith(type))
*                                          SemError("invalid cast");
*                                        else type = Types.charType;
*                                        CodeGen.CastToChar(); .)
*          | "int" ")"
*            Factor<out type>         (. if (!IsArith(type))
*                                          SemError("invalid cast");
*                                        else type = Types.intType; .)
*          | Expression<out type> ")"
*        ) .
```

Strictly speaking the above grammar departs slightly from the Java version, where the casting operator is regarded as weaker than the parentheses around an *Expression*, but in practice it makes little difference.

Various of the other productions need modification. The presence of an arithmetic operator correctly placed between character or integer operands must result in the sub-expression so formed being of integer type (and never of character type). So, for example:

```
    MultExp<out int type>             (. int type2;
                                         int op; .)
    =  Factor<out type>
       { MulOp<out op>
         Factor<out type2>            (. if (!IsArith(type) || !IsArith(type2)) {
                                           SemError("arithmetic operands needed");
                                           type = Types.noType;
                                         }
*                                        else type = Types.intType;
                                         CodeGen.BinaryOp(op); .)
       } .
```

Similarly a prefix + or - operator applied to an integer or character *Factor* creates a new factor of integer type (see full grammar for details).

The extra code generation method we need is as follows:

```
public static void CastToChar() {
// Generates code to check that TOS is within the range of the character type
  emit(PVM.i2c);
}
```

and within the `switch` statement of the `emulator` method we need:

```
case PVM.i2c:           // check convert character to integer
  if (mem[cpu.sp] < 0 || mem[cpu.sp] > maxChar) ps = badVal;
  break;
```

The interpreter has another opcode for checked storage of characters, but if the `i2c` opcodes are inserted correctly it appears that we do not really need `stoc`:

```
case PVM.stoc:          // character checked store
  tos = Pop(); adr = Pop();
  if (InBounds(adr))
    if (tos >= 0 && tos <= maxChar) mem[adr] = tos;
    else ps = badVal;
  break;
```