Appendix C
Definition of the Programming Language CPRL

Introduction

CPRL (for **C**ompiler **PR**oject **L**anguage) is a small but complete programming language with constructs similar to those found in Ada, Java, and C. CPRL was designed to be suitable for use as a project language in an advanced undergraduate or beginning graduate course on compiler design and construction. Its features illustrate many of the basic techniques and problems associated with language translation.

C.1 Lexical Considerations

General

CPRL is case sensitive. Upper-case letters and lower-case letters are considered to be distinct in all tokens, including reserved words.

White space characters (space character, tab character, and end-of-line) serve to separate tokens; otherwise they are ignored. No token can extend past an end-of-line. Spaces may not appear in any token except character and string literals.

A comment begins with two forward slashes (`//`) and extends to the end of the line.

```
temp := x;   // swap values of x and y
x := y;
y := temp;
```

CPRL does not define a maximum line length for source code files.

Identifiers

Identifiers start with a letter and contain letters and digits. An identifier must fit on a single line, and all characters of an identifier are significant.

```
identifier = letter { letter | digit } .
letter = 'A'..'Z' + 'a'..'z' .
   // letter = [A-Za-z]   equivalent regular expression character class
digit  = '0'..'9' .
   // digit = [0-9]       equivalent regular expression character class
```

Reserved Words

The following identifiers are keywords in CPRL, and they are all reserved.

```
Boolean   Char      Integer   and       array     class
const     else      enum      exit      false     for
fun       if        loop      mod       not       of
or        private   proc      protected public    read
readln    record    return    string    then      true
type      var       when      while     write     writeln
```

Note that some keywords such as `class`, `enum`, `for`, `private`, etc. are not currently used in CPRL but are reserved for possible future use. Such keywords are essentially terminal symbols in the context-free grammar that do not appear in any rule.

Literals

An integer literal consists of a sequence of `1` or more digits.

```
intLiteral = digit { digit } .
```

A Boolean literal is either "`true`" or "`false`", and both of these words are reserved.

Similar to most C-based languages, CPRL uses the backslash (`\`) as a prefix character to denote escape sequences within character and string literals. The escape sequences used by CPRL are similar to those used in other languages.

`\b`	backspace
`\t`	tab
`\n`	linefeed (a.k.a. newline)
`\f`	form feed
`\r`	carriage return
`\"`	double quote
`\'`	single quote (a.k.a. apostrophe
`\\`	backslash

A character literal is a single printable character or an escaped character enclosed by a pair of apostrophes (sometimes called single quotes). Examples include `'A'`, `'x'`, and `'\''`. A character literal is distinct from a string literal with length one. A backslash is required for a character literal containing an apostrophe (single quote). Thus, `'\''` can be used to represent a character literal consisting of a single quote, but `'''` is not valid.

For convenience, we define a printable character as Unicode character encodable in 16 bits (using UTF-16) that is not an ISO control character, although there are some 16-bit Unicode characters that are not ISO control characters that are also not printable. The Java method `isISOControl()` in class `Character` can be used to test this condition.

A string literal is a sequence of zero or more printable or escaped characters enclosed by a pair of quotation marks (double quotes). Analogous to character literals, backslashes are required whenever a string literal contains quotation marks. Examples of string literals include `"Hello, world."`, `"It's Friday!\n"`, and `"He said, \"Good morning.\""`.

Other Symbols

The following symbols serve as delimiters, operators, and special symbols in CPRL.

```
// arithmetic operators
+    -    *    /

// relational operators
=   !=   <   <=   >   >=

// assignment and other symbols
:=   (   )   [   ]   {   }   ,   :   ;   .

// special scanning symbols
EOF, unknown
```

Note that CPRL uses the reserved word "`mod`" instead of the symbol `%` as the modulus or remainder operator. Similarly, CPRL uses the reserved word "`not`" instead of the symbol "`!`" for logical negation. (But "not equal" operator is "`!=`".)

C.2 Types

CPRL is a statically-typed language.

- Every variable, constant, or expression in the language belongs to exactly one type.
- Type is a static property and can be determined by the compiler.

In general, static typing allows better error detection at compile time as well as more efficient code generation.

Predefined Scalar Types

There are three predefined scalar types in CPRL – `Boolean`, `Integer`, and `Char`.

Type `Boolean`

Type `Boolean` is treated as a predefined type with two values, `false` and `true`. It is equivalent to type `boolean` in Java, `Boolean` in Kotlin, or `bool` in C++.

Type `Integer`

Type `Integer` is a predefined type that is equivalent to type `int` in Java or `Int` in Kotlin.

Type `Char`

Type `Char` is a predefined character type that is roughly equivalent to type `char` in Java except that all characters in CPRL must be encodable in 2 bytes (16 bits) using UTF-16.

Composite (Programmer-Defined) Types

There are three composite types in CPRL – array types, string types, and record types. Composite types must be declared before they can be referenced.

Array Types

CPRL supports one dimensional array types (but arrays of arrays can be declared). An array type is defined by giving the number of elements in the array and the component (element) type.

Examples

```
type T1 = array[10] of Boolean;
type T2 = array[10] of Integer;
type T3 = array[10] of T2;
```

Array indices are integers ranging from `0` to `n-1`, where `n` is the number of elements in the array. No bounds checking is performed when accessing an element of an array.

String Types

A CPRL string has a capacity and a length. Capacity is a compile-time (static) property, whereas length is a run-time property. The capacity must be a positive integer, and for practical reasons, the capacity must be no greater than 512; i.e., 0 < capacity ≤ 512. For string variables, the length can change by assignment or by reading a string value from standard input, but length is always less than or equal to capacity. For string literals, length always equals capacity.

Examples

```
type Name = string[20];
type MonthName = string[9];
```

Similar to arrays, the individual characters of a string can be accessed using the array index notation. Indices range from `0` to `n-1`, where `n` is the capacity of the string. No bounds checking is performed when accessing a character in a string. Similar to records, the length of a string is accessed using the dot notation, as in `s.length`.

Record Types

CPRL supports record types that are similar to `structs` in C and C#. A record type is defined by specifying the fields in the record. Unlike arrays, whose elements all have the same type, the fields of a record can have different types.

Examples

```
type Point = record
  {
    x : Integer;
    y : Integer;
  };

type Month = record
  {
    name    : MonthName;
    maxDays : Integer;
  };
```

The fields of a record are accessed using the dot notation, as `p.x` or `m.name`.

C.3 Constants and Variables

Constants and variables must be declared before they can be referenced.

Constants

A constant provides a name for a literal value. Constants are introduced by declarations of the following form.

```
"const" constId ":=" literal ";"
```

The type of the constant identifier is determined by the type of the literal, which must be an integer literal, a character literal, a boolean literal, or a string literal.

Example

```
const maxIndex := 100;
```

Variables

Variables are introduced by declarations of the following form.

$$\texttt{var varId}_1\texttt{, varId}_2\texttt{, ..., varId}_n\texttt{ : typeName := value;}$$

The type name must be one of the predefined scalar types (such as `Integer`) or an identifier representing an array type, a string type, or a record type. A variable declaration can contain an optional initialization value, which must be either a literal value or a previously declared constant.

The type name cannot be a type constructor such as array constructor; i.e., the following is **not** allowed.

```
var x : array[100] of Integer;   // *** Illegal in CPRL ***
```

Examples

```
var x1, x2 : Integer;
var found : Boolean := false;

type IntArray = array[100] of Integer;
var table : IntArray;

type Months = array[13] of Month;   // January has index 1
var  months : Months;

type Name = string[20];
var name : Name;
...

writeln name[0];       // character at index 0 (first character) in name
writeln name.length;   // the length of the string
```

C.4 Operators and Expressions

Operators

The operators, in order of precedence, are as follows.

1. Boolean negation `not`
2. Multiplying operators `*  /  mod`
3. Unary sign operators `+  -`
4. Binary adding operators `+  -`
5. Relational operators `=  !=  <  <=  >  >=`
6. Logical operators `and  or`

Expressions

For binary operators, both operands must be of the same type. Objects are considered to have the same type if and only if they have the same type name. Thus, two distinct type definitions are considered different even though they may be structurally identical. This is referred to as "name equivalence" of types.

Example

```
type T1 = array[10] of Integer;
type T2 = array[10] of Integer;

var x : T1;
var y : T1;
var z : T2;
```

In the above example, x and y have the same type, but x and z do not even though they have the same structure.

Logical expressions (expressions involving logical operators "and" or "or") use short-circuit evaluation. For example, given an expression of the form "$expr_1$ and $expr_2$", the left operand ($expr_1$) is evaluated first. If the result is false, the right operand ($expr_2$) is not evaluated, and the truth value for the compound expression is considered to be false.

C.5 Statements

Assignment Statement

The assignment symbol is ":=". An assignment statement has the following form.

```
variable := expression;
```

Example 1

```
i := 2*i + 5;
```

The variable on the left and the expression on the right must have assignment compatible types. In general, CPRL uses named type equivalence, which means that two types are assignment compatible only if they share the same type name. Two composite types with identical structure but different type names are not assignment compatible.

Example 2

```
type T1 = array[10] of Integer;
type T2 = array[10] of Integer;

var x : T1;
var y : T1;
var z : T2;
...

x := y;   // allowed
x := z;   // *** Illegal in CPRL ***
```

String literals, and constants representing string literals, can be assigned to a variable of any string type provided that the length of the literal is less than or equal to the capacity of the string variable .

Example 3

```
type EmailAddress = string[30];
var emailAddr1, emailAddr2 : EmailAddress;
...
emailAddr1 := "natem@gmail.com";   // valid
emailAddr2 := "john_smith@hamptoncommunitycollege.edu";   // not valid
```

In this example, the assignment to `emailAddr2` is not valid since the length of the string literal exceeds the capacity of the string variable.

Compound Statement

A compound statement is a sequence of zero or more statements enclosed in braces "{" and "}". A compound statement can be used anywhere a single statement can be used. Compound statements are commonly used in conjunction with `if` statements and `loop` statements. For example, the body of a `loop` is exactly one statement, but it may be a compound statement.

Example

```
while i < length loop
  {
    writeln a[i];
    i := i + 1;
  }
```

compound statement

If Statement

An `if` statement in CPRL is similar to an `if` statement in Java or Kotlin except that the boolean expression is followed by the keyword "`then`" and is not enclosed in parentheses. An `if` statement can contain an optional `else` clause.

Examples

```
if a[i] = searchValue then
    found := true;

if x >= y then
    max := x;
else
    max := y;

if x < y then
  {
    // swap x and y
    temp := x;
    x := y;
    y := temp;
  }
```

Loop and Exit Statements

A `loop` statement consists of the keyword "`loop`" followed by a statement, which is often a compound statement. A `loop` statement may be preceded by an optional "`while`" clause. An `exit` statement can be used to exit the inner most loop that contains it. Note that an `exit` statement must be nested within the body of a `loop` statement; it cannot appear as a standalone statement outside of a loop.

Examples

```
while i < n loop
  {
    sum := sum + a[i];
    i := i + 1;
  }

loop
  {
    read x;
    exit when x = SIGNAL;
    process(x);
  }
```

Input/Output Statements

CPRL defines only sequential text I/O for two basic character streams – standard input and standard output. I/O is provided by `read`, `write`, and `writeln` statements. Note that `readln` is a reserved word, but it is currently not used for input in CPRL.

The `write` and `writeln` statements can have multiple expressions separated by commas. Both input and output are supported for integers, characters, and strings. Output for boolean values is supported, but it writes out an integer representation, not "`true`" or "`false`".

Examples

```
read x;     // x has type Integer
writeln "The answer is ", 2*x + 1;
```

C.6 Programs

A program has an optional list of initial declarations (`const` declarations, `var` declarations, and `type` declarations) followed by one or more subprogram declarations. One of the subprograms must be a parameterless procedure named "`main()`", which serves as the starting point for program execution.

Example

```
var x : Integer;

proc main()
  {
    read x;
    writeln "x = ", x;
  }
```

C.7 Subprograms

CPRL provides two separate forms of subprograms – procedures and functions. A procedure (similar to a `void` function in C or C++) does not return a value; it is invoked through a procedure call statement. A function must return a value and is invoked as an expression. Keywords "`proc`" and "`fun`" are used to start the declarations of procedures and functions, respectively. Recursive invocations of subprograms are allowed. Subprograms are not required to be declared before they are called. All subprogram names in a program must be distinct.

Procedures

Procedures are similar to void functions in C. Explicit `return` statements are allowed within the subprogram body, but unlike functions, a `return` statement in a procedure must **not** be followed by an expression. Procedure calls are statements.

Example (Quick Sort)

```
proc quickSort(var a : A, fromIndex : Integer, toIndex : Integer)
  {
    var i, j, pivot, temp : Integer;

    i := fromIndex;
    j := toIndex;
    pivot := a[(fromIndex + toIndex)/2];

    // partition a[fromIndex]..a[toIndex] with pivot as the dividing item
    while i <= j loop
      {
        while a[i] < pivot loop
           i := i + 1;

        while a[j] > pivot loop
           j := j - 1;
```

```
        if i <= j then
          {
            // swap a[i] and a[j]
            temp := a[i];
            a[i] := a[j];
            a[j] := temp;

            // update i and j
            i := i + 1;
            j := j - 1;
          }
      }

    if fromIndex < j then
        quickSort(a, fromIndex, j);   // sort top part

    if i < toIndex then
        quickSort(a, i, toIndex);     // sort bottom part
  }
```

Assume that we have the following declarations.

```
const arraySize := 10;
type A = array[arraySize] of Integer;
var a : A;
```

Then we could call this procedure to sort the array as follows.

```
quickSort(a, 0, arraySize - 1);
```

Functions

Functions are similar to procedures except that functions return values. Function calls are expressions. A function returns a value by executing a "`return`" statement of the following form.

```
return <expression>;
```

Example

```
fun max(x : Integer, y : Integer) : Integer
  {
    if x >= y then
        return x;
    else
        return y;
  }
```

Parameters

There are two parameter modes in CPRL – value parameters and variable parameters. Value parameters are passed by value (a.k.a. copy-in) and are the default. Variable parameters are passed by reference and must be explicitly declared using the "`var`" keyword. Unlike other types, arrays are always passed by reference to subprograms regardless of whether or not they are declared as variable parameters.

Example

```
proc inc(var x : Integer)
  {
    x := x + 1;
  }
```

Functions cannot have variable parameters; only value parameters are permitted for functions. However, arrays are always passed by reference, even for functions, where they are not declared as `var` parameters.

String literals may not be passed as actual parameters.

Return Statements

A `return` statement terminates execution of a subprogram and returns control back to the point where the subprogram was called. A `return` statement within a function must be followed by an expression whose value is returned by the function. The type of the expression must be assignment compatible with the return type of the function. A `return` statement within a procedure must not be followed by an expression; it simply returns control to the statement following the procedure call statement.

A procedure has an implied `return` statement as its last statement, and therefore most procedures will not have an explicit `return` statement. A function requires one or more `return` statements to return the function value. There is no implicit `return` statement at the end of a function.