CA4003 Compiler Construction
Assignment
Language Definition

David Sinclair

1 Overview

The language is not case sensitive. A nonterminal, X, is represented by enclosing it in angle brackets, e.g. $\langle X \rangle$. A terminal is represented without angle brackets. A **bold typeface** is used to represent terminal symbols in the language and reserved words, whereas a non-bold typeface is used for symbols that are used to group terminals and nonterminals together. Source code is kept in files with the .ccl extension, e.g. hello_world.ccl .

2 Syntax

The reserved words in the language are **var**, **const**, **return**, **integer**, **boolean**, **void**, **main**, **if**, **else**, **while** and **skip**.

The following are tokens in the language: **, ; : = { } ( ) + - ~ || && == != < <= > >=**

Integers are represented by a string of one or more digits ('0'-'9') that do not start with the digit '0', but may start with a minus sign ('-'), e.g. 123, -456.

Identifiers are represented by a string of letters, digits or underscore character ('_') beginning with a letter. Identifiers cannot be reserved words.

Comments can appear between any two tokens. There are two forms of comment: one is delimited by /* and */ and can be nested; the other begins with // and is delimited by the end of line and this type of comments may not be nested.

$$
\begin{array}{rcll}
\langle\text{program}\rangle & \models & \langle\text{decl\_list}\rangle\ \langle\text{function\_list}\rangle\ \langle\text{main}\rangle & (1)\\
\langle\text{decl\_list}\rangle & \models & (\langle\text{decl}\rangle\ \textbf{;}\ \langle\text{decl\_list}\rangle\ \mid\ \epsilon) & (2)\\
\langle\text{decl}\rangle & \models & \langle\text{var\_decl}\rangle\ \mid\ \langle\text{const\_decl}\rangle & (3)\\
\langle\text{var\_decl}\rangle & \models & \textbf{var}\ \text{identifier}\textbf{:}\langle\text{type}\rangle & (4)\\
\langle\text{const\_decl}\rangle & \models & \textbf{const}\ \text{identifier}\textbf{:}\langle\text{type}\rangle = \langle\text{expression}\rangle & (5)\\
\langle\text{function\_list}\rangle & \models & (\langle\text{function}\rangle\ \langle\text{function\_list}\rangle\ \mid\ \epsilon) & (6)\\
\langle\text{function}\rangle & \models & \langle\text{type}\rangle\ \text{identifier}\ \textbf{(}\langle\text{parameter\_list}\rangle\textbf{)} & (7)\\
 & & \textbf{\{} & \\
 & & \langle\text{decl\_list}\rangle & \\
 & & \langle\text{statement\_bock}\rangle & \\
 & & \textbf{return}\ \textbf{(}\ \langle\text{expression}\rangle\ \mid\ \epsilon\ \textbf{)}\ \textbf{;} & \\
 & & \textbf{\}} & \\
\langle\text{type}\rangle & \models & \textbf{integer}\ \mid\ \textbf{boolean}\ \mid\ \textbf{void} & (8)\\
\langle\text{parameter\_list}\rangle & \models & \langle\text{nemp\_parameter\_list}\rangle\ \mid\ \epsilon & (9)\\
\langle\text{nemp\_parameter\_list}\rangle & \models & \text{identifier:}\langle\text{type}\rangle\ \mid\ \text{identifier:}\langle\text{type}\rangle\ \textbf{,}\ \langle\text{nemp\_parameter\_list}\rangle & \\
\langle\text{main}\rangle & \models & \textbf{main}\ \textbf{\{} & (10)\\
 & & \langle\text{decl\_list}\rangle & \\
 & & \langle\text{statement\_block}\rangle & \\
 & & \textbf{\}} & \\
\langle\text{statement\_block}\rangle & \models & (\langle\text{statement}\rangle\ \langle\text{statement\_block}\rangle)\ \mid\ \epsilon & (11)\\
\langle\text{statement}\rangle & \models & \text{identifier} = \langle\text{expression}\rangle\ \textbf{;}\ \mid & (12)\\
 & & \text{identifier}\ \textbf{(}\ \langle\text{arg\_list}\rangle\ \textbf{)}\ \textbf{;}\ \mid & \\
 & & \textbf{\{}\ \langle\text{statement\_block}\rangle\ \textbf{\}}\ \mid & \\
 & & \textbf{if}\ \langle\text{condition}\rangle\ \textbf{\{}\ \langle\text{statement\_block}\rangle\ \textbf{\}}\ \textbf{else}\ \textbf{\{}\langle\text{statement\_block}\rangle\ \textbf{\}}\ \mid & \\
 & & \textbf{while}\ \langle\text{condition}\rangle\ \textbf{\{}\ \langle\text{statement\_block}\rangle\ \textbf{\}}\ \mid & \\
 & & \textbf{skip}\ \textbf{;} & \\
\langle\text{expression}\rangle & \models & \langle\text{fragment}\rangle\ \langle\text{binary\_arith\_op}\rangle\ \langle\text{fragment}\rangle\ \mid & (13)\\
 & & \textbf{(}\ \langle\text{expression}\rangle\ \textbf{)}\ \mid & \\
 & & \text{identifier}\ \textbf{(}\langle\text{arg\_list}\rangle\ \textbf{)} & \\
\langle\text{binary\_arith\_op}\rangle & \models & +\ \mid\ \textbf{-} & (14)\\
\langle\text{fragment}\rangle & \models & \text{identifier}\ \mid\ \text{number}\ \mid\ \textbf{true}\ \mid\ \textbf{false}\ \mid\ \langle\text{expression}\rangle & (15)
\end{array}
$$

$$\begin{aligned}
\langle\text{condition}\rangle &\models \sim\langle\text{condition}\rangle \mid & (16)\\
&\quad \textbf{(}\ \langle\text{condition}\rangle\ \textbf{)} \mid \\
&\quad \langle\text{expression}\rangle\ \langle\text{comp\_op}\rangle\ \langle\text{expression}\rangle \mid \\
&\quad \langle\text{condition}\rangle(\ ||\ \mid\ \textbf{\&\&}\ )\langle\text{condition}\rangle \\
\langle\text{comp\_op}\rangle &\models \textbf{==} \mid \textbf{!=} \mid < \mid <= \mid > \mid >= & (17)\\
\\
\langle\text{arg\_list}\rangle &\models \langle\text{nemp\_arg\_list}\rangle \mid \epsilon & (18)\\
\langle\text{nemp\_arg\_list}\rangle &\models \text{identifier} \mid \text{identifier}\ \textbf{,}\ \langle\text{nemp\_arg\_list}\rangle & (19)
\end{aligned}$$

3 Semantics

Declaration made outside a function (including main) are global in scope. Declarations inside a function are local in scope to that function. Function arguments are *passed-by-value*. Variables or constants cannot be declared using the void type. The skip statement does nothing.

The operators in the language are:

Operator	Arity	Description
=	binary	assignment
+	binary	arithmetic addition
-	binary	arithmetic subtraction
~	unary	logical negation
||	binary	logical disjunction (logical or)
&&	binary	logical conjunction (logical and)
==	binary	is equal to (arithmetic and logical)
!=	binary	is not equal to (arithmetic and logical)
<	binary	is less than (arithmetic)
<=	binary	is less than or equal to (arithmetic)
>	binary	is greater than (arithmetic)
>=	binary	is greater than or equal to (arithmetic)

The following table gives the precedence (from highest to lowest) and associativity of these operators.

Operator(s)	Associativity
~	right to left
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
=	right to left

4 Examples

Three versions of the simplest non-empty file demonstrating that the language is case insensitive.

```
main          Main          MAIN
{             {             {
}             }             }
```

A simple file demonstrating comments.

```
main
{
  // a simple comment
  /* a comment /* with /* several */ nested */ comments */
}
```

The simplest program that uses functions.

```
void func ()
{
  return ();
}

main
{
  func ();
}
```

A simple file demonstrating the different scopes.

```
var i:integer;

integer test_fn (x:integer)
{
  var i:integer;

  i = 2;
  return (x);
}

main
{
  var i:integer;

  i = 1;
  i = test_fn (i);
}
```

A file demonstrating the use of functions.

```
integer multiply (x:integer, y:integer)
{
  var result:integer;
  var minus_sign : boolean;

  // figure out sign of result and convert args to absolute values

  if (x < 0 && y >= 0)
  {
    minus_sign = true;
    x = -x;
  }
  else if y < 0 && x >= 0
  {
    minus_sign = true;
    y = -y;
  }
  else if (x < 0) && y < 0
  {
    minus_sign = false;
    x = -x;
    y = -y;
  }
  else
  {
    minus_sign = false;
  }
  result = 0;

  while (y > 0)
  {
    result = result + x;
    y = y - 1;
  }

  if minus_sign == true
  {
    result = -result;
  }
  else
  {
    skip;
  }

    return (result);
}

main
{
  var arg_1:integer;
  var arg_2:integer;
  var result:integer;
  const five:integer = 5;

  arg_1 = -6;
  arg_2 = five;

  result = multiply (arg_1, arg_2);
}
```