

CA4003 Compiler Construction

Assignment

Language Definition

Dr. David Sinclair

2023 – 2024

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 should be 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**, **true**, **false**, **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 or underscore character. 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{aligned}
\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 ; \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 \mathbf{var} \text{ identifier} : \langle \text{type} \rangle & (4) \\
\langle \text{const_decl} \rangle & \models \mathbf{const} \text{ identifier} : \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 } (\langle \text{parameter_list} \rangle) & (7) \\
& \{ \\
& \quad \langle \text{decl_list} \rangle \\
& \quad \langle \text{statement_block} \rangle \\
& \quad \mathbf{return} (\langle \text{expression} \rangle \mid \epsilon) ; \\
& \} \\
\langle \text{type} \rangle & \models \mathbf{integer} \mid \mathbf{boolean} \mid \mathbf{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 , \langle \text{nemp_parameter_list} \rangle & (10) \\
\langle \text{main} \rangle & \models \mathbf{main} \{ & (10) \\
& \quad \langle \text{decl_list} \rangle \\
& \quad \langle \text{statement_block} \rangle \\
& \} \\
\langle \text{statement_block} \rangle & \models (\langle \text{statement} \rangle \langle \text{statement_block} \rangle) \mid \epsilon & (11) \\
\langle \text{statement} \rangle & \models \mathbf{identifier} = \langle \text{expression} \rangle ; \mid & (12) \\
& \mathbf{identifier} (\langle \text{arg_list} \rangle) ; \mid \\
& \{ \langle \text{statement_block} \rangle \} \mid \\
& \mathbf{if} \langle \text{condition} \rangle \{ \langle \text{statement_block} \rangle \} \mathbf{else} \{ \langle \text{statement_block} \rangle \} \mid \\
& \mathbf{while} \langle \text{condition} \rangle \{ \langle \text{statement_block} \rangle \} \mid \\
& \mathbf{skip} ; \\
\langle \text{expression} \rangle & \models \langle \text{fragment} \rangle \langle \text{binary_arith_op} \rangle \langle \text{fragment} \rangle \mid & (13) \\
& (\langle \text{expression} \rangle) \mid \\
& \mathbf{identifier} (\langle \text{arg_list} \rangle) \mid \\
& \langle \text{fragment} \rangle \\
\langle \text{binary_arith_op} \rangle & \models + \mid - & (14)
\end{aligned}$$

$$\langle \text{fragment} \rangle \models \text{identifier} \mid - \text{identifier} \mid \text{number} \mid \text{true} \mid \text{false} \mid \langle \text{expression} \rangle \quad (15)$$

$$\begin{aligned} \langle \text{condition} \rangle \models & \sim \langle \text{condition} \rangle \mid \\ & (\langle \text{condition} \rangle) \mid \\ & \langle \text{expression} \rangle \langle \text{comp_op} \rangle \langle \text{expression} \rangle \mid \\ & \langle \text{condition} \rangle (\parallel \mid \&\&) \langle \text{condition} \rangle \end{aligned} \quad (16)$$

$$\langle \text{comp_op} \rangle \models == \mid != \mid < \mid <= \mid > \mid >= \quad (17)$$

$$\langle \text{arg_list} \rangle \models \langle \text{nemp_arg_list} \rangle \mid \epsilon \quad (18)$$

$$\langle \text{nemp_arg_list} \rangle \models \text{identifier} \mid \text{identifier}, \langle \text{nemp_arg_list} \rangle \quad (19)$$

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	arithmetic negation
~	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	Notes
<code>~</code>	right to left	logical negation
<code>-</code>	right to left	arithmetic negation
<code>+</code> <code>-</code>	left to right	addition & subtraction
<code><</code> <code><=</code> <code>></code> <code>>=</code>	left to right	arithmetic comparison operators
<code>==</code> <code>!=</code>	left to right	equality & inequality operators
<code>&&</code>	left to right	logical conjunction
<code> </code>	left to right	logical disjunction
<code>=</code>	right to left	assignment

4 Examples

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

<code>main</code>	<code>Main</code>	<code>MAIN</code>
<code>{</code>	<code>{</code>	<code>{</code>
<code>}</code>	<code>}</code>	<code>}</code>

A simple file demonstrating identifier names.

```
main
{
  var _time:integer;
  var start:integer;

  start = 0;
  _time = start;
}
```

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);
}

```