CA4003 Compiler Construction
Assignment
Language Definition

Dr. David Sinclair

2021 – 2022

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. 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.

⟨program⟩ ⊨ ⟨decl_list⟩ ⟨function_list⟩ ⟨main⟩ (1)

⟨decl_list⟩ ⊨ (⟨decl⟩ **;** ⟨decl_list⟩ | ϵ) (2)

⟨decl⟩ ⊨ ⟨var_decl⟩ | ⟨const_decl⟩ (3)

⟨var_decl⟩ ⊨ **var** `identifier:`⟨type⟩ (4)

⟨const_decl⟩ ⊨ **const** `identifier:`⟨type⟩ **=** ⟨expression⟩ (5)

⟨function_list⟩ ⊨ (⟨function⟩ ⟨function_list⟩ | ϵ) (6)

⟨function⟩ ⊨ ⟨type⟩ `identifier` **(**⟨parameter_list⟩**)** (7)
{
⟨decl_list⟩
⟨statement_bock⟩
return (⟨expression⟩ | ϵ **) ;**
}

⟨type⟩ ⊨ **integer** | **boolean** | **void** (8)

⟨parameter_list⟩ ⊨ ⟨nemp_parameter_list⟩ | ϵ (9)

⟨nemp_parameter_list⟩ ⊨ `identifier:`⟨type⟩ | `identifier:`⟨type⟩ **,** ⟨nemp_parameter_list⟩

⟨main⟩ ⊨ **main {** (10)
⟨decl_list⟩
⟨statement_block⟩
}

⟨statement_block⟩ ⊨ (⟨statement⟩ ⟨statement_block⟩) | ϵ (11)

⟨statement⟩ ⊨ `identifier` **=** ⟨expression⟩ **;** | (12)
`identifier` **(** ⟨arg_list⟩ **) ;** |
{ ⟨statement_block⟩ **}** |
if ⟨condition⟩ **{** ⟨statement_block⟩ **} else {**⟨statement_block⟩ **}** |
while ⟨condition⟩ **{** ⟨statement_block⟩ **}** |
skip ;

⟨expression⟩ ⊨ ⟨fragment⟩ ⟨binary_arith_op⟩ ⟨fragment⟩ | (13)
(⟨expression⟩ **)** |
`identifier` **(**⟨arg_list⟩ **)** |
⟨fragment⟩

⟨binary_arith_op⟩ ⊨ + | - (14)

⟨fragment⟩ ⊨ `identifier` | `-` `identifier` | `number` | **true** | **false** | ⟨expression⟩ (15)

⟨condition⟩ ⊨ ∼ ⟨condition⟩ | **(** ⟨condition⟩ **)** | ⟨expression⟩ ⟨comp_op⟩ ⟨expression⟩ | ⟨condition⟩(|| | **&&**)⟨condition⟩ (16)

⟨comp_op⟩ ⊨ **==** | **!=** | < | <= | > | >= (17)

⟨arg_list⟩ ⊨ ⟨nemp_arg_list⟩ | ϵ (18)

⟨nemp_arg_list⟩ ⊨ `identifier` | `identifier` **,** ⟨nemp_arg_list⟩ (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
~	right to left	logical negation
-	right to left	arithmetic negation
+ -	left to right	addition & subtraction
< <= > >=	left to right	arithmetic comparison operators
== !=	left to right	equality & inequality operators
&&	left to right	logical conjunction
||	left to right	logical disjunction
=	right to left	assignment

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);
}
```