

# CSC1098 Compiler Construction

## Assignment

### Language Definition

David Sinclair

2024-2025

## 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 `.cal` extension, e.g. `hello_world.cal`.

## 2 Syntax

The reserved words in the language are **variable**, **constant**, **return**, **int**, **bool**, **void**, **main**, **if**, **else**, **true**, **false**, **while**, **begin**, **end**, **is** and **skip**.

The following are tokens in the language:

`, ; : := ( ) + - ~ | & = != < <= > >=`

Integers are represented by a string of one or more digits ('0'-'9') and may start with a minus sign ('-'), e.g. `123`, `-456`. Unless it is the number '0', numbers may not start with leading '0's. For example, `0012` is illegal.

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

Booleans can have the value **true** or **false**.

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{variable\ identifier} : \langle \text{type} \rangle & (4) \\
\langle \text{const\_decl} \rangle &\models \mathbf{constant\ 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 \mathbf{identifier} (\langle \text{parameter\_list} \rangle) \mathbf{is} & (7) \\
&\quad \langle \text{decl\_list} \rangle \\
&\quad \mathbf{begin} \\
&\quad \langle \text{statement\_block} \rangle \\
&\quad \mathbf{return} ( \langle \text{expression} \rangle \mid \epsilon ) ; \\
&\quad \mathbf{end} \\
\langle \text{type} \rangle &\models \mathbf{int} \mid \mathbf{bool} \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 \mathbf{identifier} : \langle \text{type} \rangle \mid \mathbf{identifier} : \langle \text{type} \rangle , \langle \text{nemp\_parameter\_list} \rangle & (10) \\
\langle \text{main} \rangle &\models \mathbf{main} \\
&\quad \mathbf{begin} \\
&\quad \langle \text{decl\_list} \rangle \\
&\quad \langle \text{statement\_block} \rangle \\
&\quad \mathbf{end} \\
\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) \\
&\quad \mathbf{identifier} ( \langle \text{arg\_list} \rangle ) ; \mid \\
&\quad \mathbf{begin} \langle \text{statement\_block} \rangle \mathbf{end} \mid \\
&\quad \mathbf{if} \langle \text{condition} \rangle \mathbf{begin} \langle \text{statement\_block} \rangle \mathbf{end} \\
&\quad \mathbf{else\ begin} \langle \text{statement\_block} \rangle \mathbf{end} \mid \\
&\quad \mathbf{while} \langle \text{condition} \rangle \mathbf{begin} \langle \text{statement\_block} \rangle \mathbf{end} \mid \\
&\quad \mathbf{skip} ; \\
\langle \text{expression} \rangle &\models \langle \text{fragment} \rangle \langle \text{binary\_arith\_op} \rangle \langle \text{fragment} \rangle \mid & (13) \\
&\quad ( \langle \text{expression} \rangle ) \mid \\
&\quad \mathbf{identifier} ( \langle \text{arg\_list} \rangle ) \mid
\end{aligned}$$

$$\langle \text{binary\_arith\_op} \rangle \models + \mid - \quad (14)$$

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

$$\langle \text{condition} \rangle \models \sim \langle \text{condition} \rangle \mid ( \langle \text{condition} \rangle ) \mid \quad (16)$$

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

$$\langle \text{arg\_list} \rangle \models \langle \text{nemp\_arg\_list} \rangle \mid \epsilon \quad (18)$$

$$\langle \text{nemp\_arg\_list} \rangle \models \text{identifier} \mid \text{identifier} , \langle \text{nemp\_arg\_list} \rangle \quad (19)$$

### 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
<code>:=</code>	binary	assignment
<code>+</code>	binary	arithmetic addition
<code>-</code>	binary	arithmetic subtraction
<code>-</code>	unary	arithmetic negation
<code>~</code>	unary	logical negation
<code> </code>	binary	logical disjunction (logical or)
<code>&amp;</code>	binary	logical conjunction (logical and)
<code>=</code>	binary	is equal to (arithmetic and logical)
<code>!=</code>	binary	is not equal to (arithmetic and logical)
<code>&lt;</code>	binary	is less than (arithmetic)
<code>&lt;=</code>	binary	is less than or equal to (arithmetic)
<code>&gt;</code>	binary	is greater than (arithmetic)
<code>&gt;=</code>	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
begin	begin	begin
end	eND	end

A simple file demonstrating comments.

```
main
begin
  // a simple comment
  /* a comment /* with /* several */ nested */ comments */
end
```

The simplest program that uses functions.

```
void func () is
begin
  return ();
end
```

```

main
begin
  func ();
end

```

A simple file demonstrating the different scopes.

```

variable i:int;

int test_fn (x:int) is
  variable i:int;
begin
  i := 2;
  return (x);
end

```

```

main
begin
  variable i:int;

  i := 1;
  i := test_fn (i);
end

```

A file demonstrating the use of functions.

```

int multiply (x:int , y:int) is
  variable result:int;
  variable minus_sign : bool;
begin
  // figure out sign of result and convert args to absolute values

  if (x < 0 & y >= 0)
  begin
    minus_sign := true;
    x := -x;
  end
  else
  begin
    if y < 0 & x >= 0

```

```

begin
    minus_sign := true;
    y := -y;
end
else
begin
    if (x < 0) & y < 0
    begin
        minus_sign := false;
        x := -x;
        y := -y;
    end
    else
    begin
        minus_sign := false;
    end
    end
end
end

result := 0;

while (y > 0)
begin
    result := result + x;
    y := y - 1;
end

if minus_sign = true
begin
    result := -result;
end
else
begin
    skip;
end

return (result);
end

main

```

```
begin
  variable arg_1:int;
  variable arg_2:int;
  variable result:int;
  constant five:int := 5;

  arg_1 := -6;
  arg_2 := five;

  result := multiply (arg_1 , arg_2);
end
```