COMS W4115

Programming Languages and Translators

Event Driven State Language (EDSL)

Language Reference Manual (LRM)

Christopher D. Sargent
cds2131@columbia.edu
October 17, 2007

3. Language Manual

This manual is intended to be chapter 3 of the final report and as such, the numbering convention starts at 3. It is assumed that the user has read and understands the proposal white paper and therefore this chapter needs no introduction.

3.1 Token

A token can be whitespace, a comment, an identifier, a keyword, a literal, or an operator.

3.1.1 Whitespace

Whitespace:
‘ ‘
‘\t’
‘\n’
‘\r\n’

Whitespace consists of blanks or spaces, horizontal tabs, and newlines, or a concatenation of one or several of these. For purposes of compatibility with UNIX and Windows platforms, a newline may be either a linefeed character or a carriage return followed by a linefeed. Whitespace serves no other purpose than to separate other tokens.

3.1.2 Comment

Comment:
Traditional-Comment
End-of-Line-Comment

Traditional-Comment:
/* *Comment-End*

Comment-End:
* *Comment-End-Star*
[^*] *Comment-End*

Comment-End-Star:
/
* *Comment-End-Star*
[^/*] *Comment-End*

End-of-Line-Comment:
// *Characters-in-End-of-Line-Comment*

Characters-in-End-of-Line-Comment:
[^ '\n' '\r\n']
Characters-in-End-of-Line-Comment [^ '\n' '\r\n']

The traditional comment begins with the characters `/*` and ends with the characters `*/`. It can span multiple lines, but it does not nest with other comments.

The end-of-line comment begins with the characters`//`, and ends with the newline character or characters.

3.1.3 Identifier

Identifier:
Characters-in-Identifier

Characters-in-Identifier:
*([A-Z] | [a-z]) ([A-Z] | [a-z] | [0-9])**

An identifier begins with at least one letter and follows with a sequence of zero or more letters and digits. Identifiers are case sensitive and may have any length, subject to platform specific limitations.

3.1.3.1 Keyword

A keyword is an identifier that has special meaning within the language. Keywords should not be used except for their intended purpose. Keywords are identified within the following sections of this document by their courier font.

3.1.4 Literal

Literal:
Boolean-Literal
Integer-Literal
Real-Literal

Boolean-Literal:
`true`
`false`

Integer-Literal:
Decimal-Literal
Hexadecimal-Literal

Binary-Literal

Decimal-Literal:
[0-9]+

Hexadecimal-Literal:
0 (x|X) ([A-F] | [a-f] | [0-9])*

Binary-Literal:
0 (y|Y) [0-1]*

Real-Literal:
'.' [0-9]+ *Exponent*?
[0-9]+ ('.' [0-9]+ *Exponent*? | *Exponent*)

Exponent:
(e|E) ('+' | '-')? [0-9]+

A Boolean literal has one of two values, either `true` or `false`.

An integer literal can be a decimal, hexadecimal, or binary number. The prefix is necessary for an LL(2) parser to be able to distinguish the three different kinds of integer.

A real literal "consists of an integer part, a decimal point, a fraction part, an e or E, an optionally signed integer exponent. … The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing."[1]

3.1.5 Operator

The function of the operators may depend on their operands. See section 3.2 for their usage within expressions.

3.2 Type and Type Conversion

There are four basic types. The first three types are simple value storage types. The last is unique in that it stores a value, but it also operates on that value.

Like the Boolean literal, a Boolean variable takes either the `true` or `false` value.

[1] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language Second Edition (Murray Hill: Prentice-Hall, 1988) 194.

The integer variable takes a 32-bit signed integer. The minimum value of an integer is `-2147483648` and the maximum is `2147483647`.

The real variable takes a 64-bit double-precision real. Its size and representation are governed by ANSI/IEEE Std. 754-1985.

A timer has the same size and representation as a positive real, but it is constantly decremented until its value reaches zero. Its units are seconds relative to wall-clock time.

Automatic type conversion may occur as a result of certain operators and their required operands. Automatic or implicit conversion is the only allowed type conversion mechanism.

3.2.1 Boolean and Integer

A `bool` may be converted to an `int` without loss. If its value is `false`, then its integer representation will be `0`. If its value is true, then its integer representation will be `1`.

An `int` may be converted to a `bool` with data loss. If its value is `0`, then its Boolean representation will be `false`. If its value is not `0` (either a positive or negative value), then its Boolean representation will be `true`.

3.2.2 Integer and Real

An `int` may be converted to a `real` without loss. Its value is exactly the same, with the exception that it has a fractional component equal to zero.

A `real` may be converted to an `int` with data loss. Not only will its fractional component be truncated towards zero, but if the magnitude of its whole number component exceeds the minimum or maximum for an integer, it will be reduced to that value.

3.2.3 Real and Timer

A `timer` may not take a value that is not an acceptable `real` value. In addition, a timer may not store a negative value. If a timer is assigned a negative value, its sign will be inverted.

The value that a `timer` currently has may be converted to a `real` without loss.

3.3 Expression

The precedence of the expressions is in order that the expressions are in, in this section.

3.3.1 Primary Expression

primary-expression:
  Identifier
  Literal
  (*expression*)

The primary expressions consist of identifiers, literals, and expressions in parentheses.

Identifier

See section 3.1.3.

Literal

See section 3.1.4.

(*expression*)

A parenthesized expression is equivalent to the same expression without the parentheses.

3.3.2 Unary Expression

unary-expression:
  primary-expression
  -*unary-expression*
  !*unary-expression*

The unary operators: - and ! evaluate from right-to-left.

-*unary-expression*

For the ‘-‘ or ‘negative’ operator, the result of the evaluation is the negative of the operand.

!*unary-expression*

For the ‘!’ or logical ‘negation’ operator, the result of the evaluation is `true` if the operand is `false` and `false` if the operand is `true`.

3.3.3 Multiplication, Division, or Remainder Expression

multiplication-expression:
 unary-expression
 multiplication-expression * *unary-expression*
 multiplication-expression / *unary-expression*
 multiplication-expression % *unary-expression*

The multiplication, division, and remainder operators: *, /, and % evaluate from left-to-right.

multiplication-expression * *unary-expression*

For the '*' or 'multiplication' operator, the result of the first operand is multiplied by the result of the second operand.

multiplication-expression / *unary-expression*

For the '/' or 'division' operator, the result of the first operand is divided by the result of the second operand. If the second operand is zero, the result is undefined.

multiplication-expression % *unary-expression*

For the '%' or 'remainder' operator, the result is the integer remainder of the division of the first operand by the second operand. "If both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor."[2]

3.3.3.1 *expression* * *expression*

The * operator indicates multiplication. The result is an `int` if both operands are `int` data objects. If one is an `int` and the other a `real`, then the former is converted to a `real`, and the result is a `real`. The timer is treated as a `real` for this operator. No other combinations are permitted.

3.3.3.2 *expression* / *expression*

The / operator indicates division. The result is an `int` if both operands are `int` data objects. If one is an `int` and the other a `real`, then the former is converted to a `real`, and the result is a

[2] Kernighan and Ritchie 205.

`real`. The timer is treated as a `real` for this operator. No other combinations are permitted.

3.3.3.3 *expression* % *expression*

The % operator results in the remainder of the division of the first operand by the second. Both operands must be `int` types and the result is an `int`. The result has the same sign as the dividend.

3.3.4 Addition or Subtraction Expression

addition-expression:
multiplication-expression
addition-expression + *multiplication-expression*
addition-expression - *multiplication-expression*

The addition and subtraction operators: + and – group from left-to-right.

addition-expression + *multiplication-expression*

For the ‘+’ or ‘addition’ operator, the result of the first operand is added to the result of the second operand.

addition-expression - *multiplication-expression*

For the ‘-’ or ‘subtraction’ operator, the result of the second operand is subtracted from the result of the first operand.

3.3.5 Shift Expression

shift-expression:
addition-expression
shift-expression << *addition-expression*
shift-expression >> *addition-expression*

The two shift operators: << and >> group from left-to-right.

shift-expression << *addition-expression*

For the ‘<<’ or ‘shift left’ operator, the result of the first operand is left-shifted by the number of bits given in the result of the second operand. Its operands must evaluate to `int` or `bool` values. Values that are not are automatically cast to the `int` type.

shift-expression >> *addition-expression*

For the ‘>>’ or ‘shift right’ operator, the result of the first operand is right-shifted by the number of bits given in the result of the second operand. Its operands must evaluate to `int` or `bool` values. Values that are not are automatically cast to the `int` type.

3.3.6 Inequality Expression

inequality-expression:
  shift-expression
  inequality-expression < *shift-expression*
  inequality-expression > *shift-expression*
  inequality-expression <= *shift-expression*
  inequality-expression >= *shift-expression*

The four inequality operators: <, >, <=, >= group from left-to-right.

inequality-expression < *shift-expression*

For the ‘<’ or ‘less than’ operator, if the first operand is less than the second operand, then the expression yields `true`, if not, it yields `false`.

inequality-expression > *shift-expression*

For the ‘>’ or ‘greater than’ operator, if the first operand is greater than the second operand, then the expression yields `true`, if not, it yields `false`.

inequality-expression <= *shift-expression*

For the ‘<=’ or ‘less than or equal to’ operator, if the first operand is less than or equal to the second operand, then the expression yields `true`, if not, it yields `false`.

inequality-expression >= *shift-expression*

For the ‘>=’ or ‘greater than or equal to’ operator, if the first operand is greater than or equal to the second operand, then the expression yields `true`, if not, it yields `false`.

3.3.7 Equality Expression

equality-expression:
  inequality-expression
  equality-expression == *inequality-expression*

equality-expression != *inequality-expression*

The two equality operators: == and != group from left-to-right.

equality-expression == *inequality-expression*

For the '==' or 'equal to' operator, if the first operand is equal to the second operand, then the expression yields `true`, if not, it yields `false`.

equality-expression != *inequality-expression*

For the '!=' or 'not equal to' operator, if the first operand is not equal to the second operand, then the expression yields `true`, if not, it yields `false`.

3.3.8 Bitwise AND Expression

bitwise-AND-expression:
equality-expression
bitwise-AND-expression & *equality-expression*

The bitwise 'and' operator evaluates from left-to-right. The result is a bitwise and function of the operands. Its operands must evaluate to `int` or `bool` values. Values that are not are automatically cast to the `int` type.

3.3.9 Bitwise XOR Expression

bitwise-XOR-expression:
bitwise-AND-expression
bitwise-XOR-expression ^ *bitwise-AND-expression*

The bitwise 'exclusive or' operator evaluates from left-to-right. The result is a bitwise exclusive or function of the operands. Its operands must evaluate to `int` or `bool` values. Values that are not are automatically cast to the `int` type.

3.3.10 Bitwise OR Expression

bitwise-OR-expression:
bitwise-XOR-expression
bitwise-OR-expression | *bitwise-XOR-expression*

The bitwise 'inclusive or' operator evaluates from left-to-right. The result is a bitwise inclusive or function of the operands. Its operands must evaluate to `int` or `bool` values. Values that are not are automatically cast to the `int` type.

3.3.11 Logical AND Expression

logical-AND-expression:
  bitwise-OR-expression
  logical-AND-expression && bitwise-OR-expression

The logical ‘and’ operator evaluates from left-to-right. The first operand is evaluated completely, and then the second operand is evaluated completely. If the result of the first evaluation is `true` and the second evaluation is `true`, then the result of the expression is `true`. Otherwise, the result of the expression is `false`.

3.3.12 Logical XOR Expression

logical-XOR-expression:
  logical-AND-expression
  logical-XOR-expression ^^ *logical-AND-expression*

The logical ‘exclusive or’ operator evaluates from left-to-right. The first operand is evaluated completely, and then the second operand is evaluated completely. If the result of the first evaluation is `true` and the second evaluation is `false`, or the first evaluation is `false` and the second evaluation is `true`; then the result of the expression is `true`. Otherwise, the result of the expression is `false`.

3.3.13 Logical OR Expression

conditional-expression:
  logical-XOR-expression
  conditional-expression || *logical-XOR-expression*

The logical ‘inclusive or’ operator evaluates from left-to-right. The first operand is evaluated, and if the result of its first evaluation is `true`, then the value of the expression is true. If the result is `false`, the second operand is evaluated and the expression is equal to its result.

3.3.14 Assignment Expression

expression:
  conditional-expression
  identifier = expression

The assignment operator evaluates from right-to-left and requires a modifiable variable as its left operand. The value of expression replaces the value of the variable referred to by the variable.

3.4 Variable Declaration

declaration:
  Linkage Type Identifier

Linkage:
```
internal
input
output
inout
```

Type:
```
bool
int
real
timer
```

The scope of a variable begins at the end of its declaration and persists to the end of the program in which it appears. Variables have the same lifetime as the running program. A variable declaration is associated with an identifier and has two configurable attributes: linkage and type.

There are two types of linkage: `internal` and external and three types of external linkage: `input`, `output`, and `inout`. The `inout` linkage is the only bi-directional external linkage type.

There are four variable types. See section 3.2.

3.5 Statement

statement:
  labeled-statement
  expression-statement
  compound-statement
  selection-statement
  jump-statement

Unlike expressions, statements do not have values.

3.5.1 Labeled Statement

labeled-statement:

Identifier : statement

A label with an identifier declares that identifier. This identifier then becomes the target of a `goto`.

3.5.2 Expression Statement

expression-statement:
expression$_{opt}$;

Each expression statement is followed by a semi-colon. For more information on expressions, see section 3.3.

3.5.3 Compound Statement

compound-statement:
{ statement-list$_{opt}$ }

statement-list
statement
statement-list statement

The compound statement is a list of zero or more statements of any kind.

3.5.4 Selection Statement

selection-statement:
`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

In both forms of the `if` statement, the expression, which must have arithmetic type, is evaluated …. If it is `true`, the first substatement is executed. In the second form, the second substatement is executed if the expression is `false`. "The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if` at the same block nesting level."[3]

3.5.5 Jump Statement

jump-statement:
`goto` *Identifier* ;

[3] Kernighan and Ritchie 223.

The identifier must be a label. Control transfers to the labeled statement.

3.6 Program Declaration

program-declaration:
    declaration-list$_{opt}$ *compound-statement*

declaration-list:
    declaration
    declaration-list declaration

The entire program consists of an optional list of variable declarations followed by a compound statement, which is a list of statements surrounded by braces.

3.7 Grammar

The grammar is a concatenated listing of the entire grammar from the previous sections. The grammar listing progresses from lower precedence to higher precedence.

program-declaration:
    declaration-list$_{opt}$ *compound-statement*

declaration-list:
    declaration
    declaration-list declaration

declaration:
    Linkage Type Identifier

Linkage:
    `internal`
    `input`
    `output`
    `inout`

Type:
    `bool`
    `int`
    `real`
    `timer`

statement:
    labeled-statement
    expression-statement

compound-statement
selection-statement
jump-statement

labeled-statement:
Identifier : statement

expression-statement:
expression$_{opt}$ *;*

compound-statement:
{ statement-list$_{opt}$ *}*

statement-list:
statement
statement-list statement

selection-statement:
`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

jump-statement:
`goto` *Identifier* ;

expression:
conditional-expression
identifier = expression

conditional-expression:
logical-XOR-expression
conditional-expression || logical-XOR-expression

logical-XOR-expression:
logical-AND-expression
logical-XOR-expression ^^ logical-AND-expression

logical-AND-expression:
bitwise-OR-expression
logical-AND-expression && bitwise-OR-expression

bitwise-OR-expression:
bitwise-XOR-expression
bitwise-OR-expression | bitwise-XOR-expression

bitwise-XOR-expression:
bitwise-AND-expression

bitwise-XOR-expression ^ *bitwise-AND-expression*

bitwise-AND-expression:
equality-expression
bitwise-AND-expression & *equality-expression*

equality-expression:
inequality-expression
equality-expression == *inequality-expression*
equality-expression != *inequality-expression*

inequality-expression:
shift-expression
inequality-expression < *shift-expression*
inequality-expression > *shift-expression*
inequality-expression <= *shift-expression*
inequality-expression >= *shift-expression*

shift-expression:
addition-expression
shift-expression << *addition-expression*
shift-expression >> *addition-expression*

addition-expression:
multiplication-expression
addition-expression + *multiplication-expression*
addition-expression - *multiplication-expression*

multiplication-expression:
unary-expression
multiplication-expression * *unary-expression*
multiplication-expression / *unary-expression*
multiplication-expression % *unary-expression*

unary-expression:
primary-expression
-unary-expression
!*unary-expression*

primary-expression:
Identifier
Literal
(*expression*)

Literal:

Boolean-Literal
Integer-Literal
Real-Literal

Boolean-Literal:
`true`
`false`

Integer-Literal:
Decimal-Literal
Hexadecimal-Literal
Binary-Literal

Decimal-Literal:
[0-9]+

Hexadecimal-Literal:
0 (x|X) ([A-F] | [a-f] | [0-9])*

Binary-Literal:
0 (y|Y) [0-1]*

Real-Literal:
‘.’ [0-9]+ _Exponent_?
[0-9]+ (‘.’ [0-9]+ _Exponent_? | _Exponent_)

Exponent:
(e|E) (‘+’ | ‘-‘)? [0-9]+

Identifier:
Characters-in-Identifier

Characters-in-Identifier:
_([A-Z] | [a-z]) ([A-Z] | [a-z] | [0-9])*_

Comment:
Traditional-Comment
End-of-Line-Comment

Traditional-Comment:
/* _Comment-End_

Comment-End:
* _Comment-End-Star_
[^*] _Comment-End_

Comment-End-Star:
/
* *Comment-End-Star*
[^/*] *Comment-End*

End-of-Line-Comment:
// *Characters-in-End-of-Line-Comment*

Characters-in-End-of-Line-Comment:
[^ ‘\n’ ‘\r\n’]
Characters-in-End-of-Line-Comment [^ ‘\n’ ‘\r\n’]

Whitespace:
‘ ‘
‘\t’
‘\n’
‘\r\n’

3.8 Bibliography

The bibliography is not included within this chapter. It will be included within the final report. For now, the foot notes should serve as the bibliography for this chapter.