TingStim

[Language Reference Manual version 1.0]

COMS W4115 [Spring 2006 CVN]
Programming Languages and Translators

By Alvin Ting
at2337@columbia.edu
2006/02/26

TingStim:

Stimulator / Interface Test Driver

1 Language Reference Manual

1.1 Lexical Conventions

1.1.1 Line Terminators

Input characters are divided into lines by line terminators. Lines are terminated by the standard ASCII characters `CR`, or `LF`, or `CR LF`. The two characters `CR` immediately followed by `LF` are counted as one line terminator, not two.

LineTerminator:
the ASCII `LF` character, also known as "linefeed"
the ASCII `CR` character, also known as "carriage-return"
the ASCII `CR` character followed by the ASCII `LF` character

InputCharacter:
Is Not *LineTerminator*

1.1.2 Comments

Comments are textual descriptions and are ignored by the compiler. There are two kinds of comments:

/ text */*	Multi-line comment: All the text from the ASCII characters `/*` to the ASCII characters `*/` is ignored.
// text	Single-line comment: All text from the ASCII characters `//` to the end of the line is ignored.

Comment:
MultiLineComment
SingleLineComment

Comments do not nest. The ASCII characters `/*` and `*/` have no special meaning when preceded with the ASCII characters `//`. In turn, the ASCII characters // has no special meaning in comments that begin with `/*`.

1.1.3 White Space

White space is defined as the ASCII space, horizontal tab, form feed characters, and line terminators. White space will be ignored when used to terminate a comment (a line terminator).

WhiteSpace:

> the ASCII `SP` character, also known as “space”
> the ASCII `HT` character, also known as “horizontal tab”
> the ASCII `FF` character, also known as “form feed”
> *LineTerminator*

1.1.4 Tokens

Tokens are subdivided into 4 classifications: identifiers, keywords, literals, separators, operators, and patterns. Tokens must be separated by white space.

Token:

> *Identifier*
> *Keyword*
> *Literals*
> *Separator*
> *Operator*
> *Pattern*

1.1.4.1 Identifiers

An identifier consists of letter, digits and underscores “_”. The first character of an identifier should be a letter or underscore. Upper and lower case letters are considered different.

1.1.4.2 Keywords

The following sequences of characters, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers:

```
close       interact   true
catch       if         false
disp        open
else        send
expect      wait
exit        while
```

1.1.4.3 Literals

Literal:

IntegerLiteral
StringLiteral
BooleanLiteral
CharacterLiteral

1.1.4.3.1 Integer Literal

Integer literals are divided into 4 classifications relative to the base of the number system: decimal, hexadecimal, octal, or binary.

IntegerLiteral:

DecimalLiteral
HexIntegerLiteral
OctalIntegerLiteral
BinaryIntegerLiteral

DecimalLiteral:

DecimalNumeral

HexIntegerLiteral:

ASCII character `x` *HexNumeral*

OctalIntegerLiteral:

ASCII character `o` *OctalNumeral*

BinaryIntegerLiteral:

ASCII character `b` *BinaryNumeral*

DecialNumeral:

(0-9) one or more times

HexNumeral:

(0..9) one or (A-F) one or more times

OctalNumeral:

0..7 one or more times

BinaryNumeral:

(0 | 1) one or more times

For example,

dec: 10000214
hex: xDEAD

octal: o077
binary: b01010101

1.1.4.3.2 String Literal

A string is a sequence of characters enclosed by ASCII double quotes “ ”.

StringLiteral:
“ *StringCharacters* “

StringCharacters:
StringCharacter
StringCharacters StringCharacter

StringCharacter:
InputCharacter but not “

1.1.4.3.3 Boolean Literal

The boolean type has two types, represented by the literals true and false, formed from ASCII letters.

BooleanLiteral:
true | false

1.1.4.3.4 Character Literal

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes ‘ ‘.

CharacterLiteral:
‘ SingleCharacter ‘
‘ EscapeSequence ‘

SingleCharacter:
InputCharacter but not ‘ or \

1.1.4.4 Separators

The following three ASCII characters are separators (punctuators):

```
{   }   ;   /
```

1.1.4.5 Operators

`=`   `==`   `!=`

The ASCII character = is the assignment operator. Two consecutive ASCII = characters is the equal-to operator. An ASCII character `!` followed by the ASCII character = is the not equal-to operator.

1.1.4.6 Pattern

A pattern is everything enclosed between an ASCII character / and ASCII character /

Where pattern can consist of combinations of the following:

regex	**English**
`S`	literal S
`\*`	literal *
`\\`	literal \
`\″`	literal “
`\′`	literal ‘
`\(`	literal (
`\)`	literal)
`^`	beginning of string
`$`	end of string
`[a-z]`	any character in the range a to z
`[^a-z]`	any character not in the range a to z
`[abc]`	a, b, or c
`.`	any character
`X*`	X character zero or more times
`X+`	X character one or more times
`X?`	X character once or none at all
`X{n}`	X character exactly n times
`X{n,}`	X character at least n times
`X{n,m}`	X character at least n but not more than m times
`( )`	Capturing group, multiple characters are treated as a single unit

For example: `/^[a-z]+z{1,3}/` is the pattern to find a string which begins with a lowercase letter followed by none or more lowercase letters, and then anywhere from one or three consecutive z.

1.2 Expressions

Expressions consist of identifiers, literals, equality, and pattern expressions.

1.2.1 Identifier

An identifier itself is a left-value expression. It will be evaluated to some value bounded to this identifier.

1.2.2 Literal

A literal is a right-value expression. It will be evaluated to the literal itself.

1.2.3 Equality

The equality operators are syntactically left-associative (they group left-to-right).

EqualityExpression:
    EqualityExpression == RelationalExpression
    EqualityExpression != RelationalExpression

The == (equal to) and the != (not equal to) operators may be used to compare two operands that of the same type. If both operands are some form of an integer literal, then both operands are first converted into decimal form prior to making the comparison. All other cases result in a compile-time error.

1.2.4 Pattern

A regular expression is a right-value expression. It will be evaluated to the pattern itself.

PatternExpression:
    Pattern

1.3 Declarations

Variables are implicitly declared upon assignment.

1.4 Statements

Statements are the basic elements of a scenario. A sequence of statements will be executed sequentially, unless flow-control statements indicate otherwise.

Statement:
    ExpressionStatement

AssignmentStatement
ConditionalStatement
LoopStatement
NullStatement
DisplayStatement
CommunicationsStatement
ExitStatement

1.4.1 Expression

Expressions may be used as statements by following them with a semicolon.

Syntax:

```
<expr>;
```

Constraints:

- none

1.4.2 Assignment

The assignment operator = is syntactically right-associative (they group right-to-left). The value of the left operand is set to the value of the right operand.

Syntax:

```
<identifier-expr>  = <literal-expr>;
<identifier-expr>  = <pattern-expr>;
<identifier-expr1> = <identifier-expr2>;
```

Constraints:

- In all cases, the left operand must be an identifier.
- The right operand may be a literal, a different identifier or a pattern.

1.4.3 Conditional

Conditionals are done with if (expression) statement else statement. If the equality expression returns true, the first statement is executed, otherwise the second statement is executed.

Syntax:

```
if ( <equality-expr> )
{
      <statement-list>
}
else
{
      <statement-list>
};
```

Constraints:

- Every “if” must have a corresponding “else.”

1.4.4 Loop

The check is done before the execution of the statement, and will be repeated as long as the test expression is true.

Syntax:
```
while ( <equality-expr> )
{
     <statement-list>
};
```
Constraints:
- none

1.4.5 Null

A null statement has no programmatic function and is stated only for completeness.

Syntax:
```
;
```
Constraints:
- none

1.4.6 Display

The display statement is used to print the value of an identifier or literal to the standard output.

Syntax:
```
disp <identifier-expr>;
disp <literal-expr>;
```
Constraints:
- none

1.4.7 Wait

The wait statement pauses scenario execution for the designated period of time.

Syntax:
```
wait <time_milliseconds>;
```

where

    <time_milliseconds>:
        decimal number

Constraints:

- Time is limited to the largest size integer.

1.4.8 Communications

1.4.8.1 Send

The send statement writes the designated literal(s) to the communication output. In the form which takes a list of integer literals, the literals are concatenated to form one numerical value prior to sending. Integer overflow is handled by setting value to the max value. The following are all valid send statements:

```
send 2002;
send b0101 b0010 b0001 b0001; // integer = 21,009
send x001A xFFFF;             // integer = 1,769,471
send o001 o123 o327;          // integer = 304,855
send "Hello, world\n";
```

Syntax:

```
      send <decimal-literal>;
      send <other-integer-literal-list>;
      send <string-literal>;
```

where

```
      <other-integer-literal-list>:
            HexIntegerLiteral
            OctalIntegerLiteral
            BinaryIntegerLiteral
```

Constraints:

- The subsequent integer literals in the list must be of the same type as the first literal specified.
- The order of list integer literals is such that the first literal contains the most significant bits and the last literal contains the least significant bits.
- The maximum integer value is equivalent to java.lang.Integer.MAX_VALUE.

1.4.8.2 Expect

The expect statement will block on reading from the current communication input. Once an input unit has been read, it will compare the input value with one or a series of valid expressions. When the first valid expression is found, the actions associated with it will be executed and the statement is considered complete.

Syntax:

```
      expect <valid-expr> { <action_list> };

      expect
      {
            <valid-expr> { <action_list> }
            <valid-expr> { <action_list> }
            …
      };
```

where

```
      <valid-expr>:
            DecimalLiteral
            <other-integer-literal-list>
            StringLiteral
            CharacterLiteral
            Pattern

      <action_list>:
            Send
            Interact
            Wait
            Loop
            Display
            Assignment
            Conditional
```

Constraints:

- Expect statements do not nest.
- <other-integer-literal-list> is the same as that defined in send statement.
- Read timeouts are currently not implemented or handled.

1.4.8.3 Catch

The catch statement is used to handle the case when unexpected input is received.

Syntax:

```
      expect ValidExpression {    <action_list> };
      catch {    <action_list> };

      expect {
            ValidExpression { <action_list> }
            ValidExpression { <action_list> }
            …
      }
      catch {    <action_list>    };
```

Constraints:

- <action_list> is same as those in expect statement.
- No other statements may go between an expect and catch statement.
- A catch statement without a proceeding expect statement is invalid.

1.4.8.4 Interact

The interact statement is used to enable the operator to alter the flow of the scenario during execution. It will block until the operator enters a keystroke. The resulting keystroke is compared with the series of expected character literals. Once a matching literal is found, the associated list of actions is executed and the statement is considered complete. If no matching literal is found then an alert will be displayed and the scenario simply continues execution.

Syntax:
```
      interact CharacterLiteral { <action_list> };

      interact
      {
            CharacterLiteral { <action_list> }
            CharacterLiteral { <action_list> }
            …
      };
```
Constraints:

- <action_list> is same as those in expect statement.

1.4.8.5 Open

The open statement opens the specified communication link. Only the first open statement is executed. Any subsequent open statements are invalid and are ignored.

Syntax:
```
      open type=<type_option>;
```
where

<type_option>:

"RS232"

"Ethernet" (unimplemented)

Constraints:

- none

1.4.8.6 Close

The close statement closes the current communication link. It is not necessary to include the close statement and is included only for completeness.

Syntax:
```
      close;
```

Constraints:

- none

1.4.9 Exit

The exit statement will terminate the scenario and no other statements are executed.

Syntax:

```
exit;
```

Constraints:

- none

1.5 Scope Rules

1.5.1 Lexical scope

This language is open scope and has only a global namespace.

1.6 Example

```
/* This line is a comment */
disp "Initializing Communications...";
open type="RS232";

// Display options
disp “Press enter to continue”
disp “Press 0 to exit”
interact {
     ‘\n’  { disp "Go!";    }
     ‘0’   { exit;          }
};

// Delay for 10ms
wait 10;

// Send stuff
send b0001 b0101 b1111 b0101;
// Expect flipped bits
expect {
     b1110 b1010 b0000 b1010 { disp “It worked!”; }
     xDEAD {
          disp “something bad happened!”;
          exit;
```

```
    }
    “nothing” { disp “This is a test”; }
}
catch { disp “I didn’t expect that!” }

close;
exit;
```