Statement

Motivation, Applicability, and Indications for Use

The programming type Statement is used to represent the abstract syntax tree (AST) of a (nested) statement in the BL language. Abstract syntax trees are common representations of programs used in language processing tools. They provide the necessary information about the source code needed to compile it or interpret it, while dispensing with the “concrete syntax” details.

Here is an example of a BL statement:

```bl
WHILE next-is-not-empty DO
  IF next-is-enemy THEN
    infect
  ELSE
    turnleft
  END IF
END WHILE
```

In Figure 1 you can see picture of a possible corresponding abstract syntax tree.

```
while
    
  next-is-not-empty
    
  if-else
       
    next-is-enemy
    infect
    turnleft
```

Figure 1 — An abstract syntax tree

As you can see, the AST is simply a different representation of the BL statement. The root item tells us that we are looking at a WHILE statement. The first child of the root shows the test condition for the while (`next-is-not-empty`), and the second child of the root is the AST for the nested statement. This happens to be an IF-ELSE statement, as indicated by the root of the second child of WHILE. The three children of the IF-THEN item are, from left to right, the IF-ELSE test condition (`next-is-enemy`), the AST for the THEN part of the IF-ELSE (`infect`), and the AST for the ELSE part of the IF-THEN (`turnleft`).

The structure of abstract syntax trees is not unique. In Figure 2 we show an alternative AST for the same statement.
First, compare the AST in Figure 2 with the BL statement above, and convince
yourself that the AST still is an exact representation of the statement (i.e., it
contains enough information to unequivocally allow us to reconstruct the statement
from the AST). Then, comparing the ASTs in Figures 1 and 2, we can see that there are two main differences:

1. In Figure 2, the test conditions for the WHILE and IF-ELSE statements have been attached to their corresponding statement items in the tree (i.e., next-is-not-empty to WHILE and next-is-enemy to IF-ELSE);

2. In Figure 2, a new item, labeled block, has been introduced in three places in
the AST. The block item allows us to group together a string of statements
nested inside one of BL’s control structures (WHILE, IF, and IF-ELSE).

Here is another example that shows why blocks are useful. Consider the BL
statement:

```
IF next-is-enemy THEN
  infect
  turnleft
ELSE
  IF next-is-wall THEN
    turnright
    turnright
  END IF
  infect
END IF
```

And in Figure 3 the corresponding AST.
Without the block items in the tree, the root IF-ELSE would have 4 children (infect, turnleft, if, infect), and there would be no indication of which of these statements are in the THEN part and which are in the ELSE part.

Related Components

- **Program** — a type representing an abstract description of a BL program, and that uses **Statement** to represent both the body of the program and the bodies of the declared instructions.

Component Family Members

Abstract Components

- **Statement_Type** — the programming type of interest with the operations below
  - *Add_To_Block*
  - *Remove_From_Block*
  - *Length_Of_Block*
  - *Compose_If*
  - *Decompose_If*
  - *Compose_If_Else*
• Decompose_If_Else
• Compose_While
• Decompose_While
• Compose_Call
• Decompose_Call
• Kind

Concrete Components

• Statement_Kernel_1a_C — This is a checking implementation of Statement_Kernel in which the execution time for each of the operations constructor, Length_Of_Block, Compose_If, Decompose_If, Compose_If_Else, Decompose_If_Else, Compose_While, Decompose_While, Compose_Call, Decompose_Call, and the Kind is constant, while the execution time for Add_To_Block and Remove_From_Block is proportional to the number of statements in the block, and the execution time for the destructor is proportional to the overall size of the statement. All objects of this type have the interface of Statement_Kernel, with the concrete template name Statement_Kernel_1a_C substituted for the abstract template name Statement_Kernel.

To bring this component into the context you write:

```cpp
#include "CI/Statement/Kernel_1a_C.h"
```

Component Coupling Diagram
// /*------------------------------*/
// | Abstract Template : Statement_Kernel
//="/------------------------------*/

#ifndef AT_STATEMENT_KERNEL
#define AT_STATEMENT_KERNEL 1

 IService

 enumeration Kind
 {
     BLOCK,
     IF,
     IF_ELSE,
     WHILE,
     CALL
 };

 enumeration Condition
 {
     NEXT_IS_EMPTY,
     NEXT_IS_NOT_EMPTY,
     NEXT_IS_WALL,
     NEXT_IS_NOT_WALL,
     NEXT_IS_FRIEND,
     NEXT_IS_NOT_FRIEND,
     NEXT_IS_ENEMY,
     NEXT_IS_NOT_ENEMY,
     RANDOM,
     TRUE
 };

 abstract_template <
     concrete_instance class Nested_Statement_Type
     /*!
         implements
             abstract_instance Statement_Kernel <
                 Nested_Statement_Type
         >
     */
class Statement_Kernel
{
    public:
        
    /*!
       math definition IS_IDENTIFIER (i: string of character) : boolean is
       i = empty_string or
       (there exists c: character, s: string of character
        (i = <c> * s and
         c is in { 'a'..'z', 'A'..'Z' } and
         for all x: character where (x is in elements (s))
          (x is in { 'a'..'z', 'A'..'Z', '0'..'9', '-'}) and
         i is not in { "PROGRAM", "IS", "INSTRUCTION", "WHILE", "DO",
                      "IF", "THEN", "ELSE", "BEGIN", "END",
                      "next-is-empty", "next-is-not-empty",
                      "next-is-wall", "next-is-not-wall",
                      "next-is-friend", "next-is-not-friend",
                      "next-is-enemy", "next-is-not-enemy",
                      "random", "true"})
       math subtype IDENTIFIER is string of character
       exemplar i
       constraint
       IS_IDENTIFIER (i)
    
    math definition IS_CONDITION (c: integer) : boolean is
    c is in Condition

    math definition IS_STATEMENT_LABEL (kind: integer test: integer instruction: string of character) : boolean is
    (kind = BLOCK and test = TRUE and instruction = empty_string) or
    ((kind = IF or kind = IF_ELSE or kind = WHILE) and instruction = empty_string) or
    (kind = CALL and test = TRUE and instruction /= empty_string)

    math subtype STATEMENT_LABEL is (kind: Kind test: Condition instruction: IDENTIFIER)
        exemplar n
        constraint
math subtype STATEMENT is tree of STATEMENT_LABEL
exemplar s
constraint
  IS_LEGAL_STATEMENT (s)

math definition IS_LEGAL_STATEMENT (t: tree of STATEMENT_LABEL)
  : boolean satisfies
there exists label: STATEMENT_LABEL,
  nested_stmts: string of tree of STATEMENT_LABEL
(t = compose (label, nested_stmts) and
  for all x: tree of STATEMENT_LABEL
    where (x is in elements (nested_stmts))
      (IS_LEGAL_STATEMENT (x)) and
    if label.kind = BLOCK
      then
        for all x: tree of STATEMENT_LABEL
          where (x is in elements (nested_stmts))
            (root (x).kind /= BLOCK)
        else if label.kind = IF or label.kind = WHILE
          then
            |nested_stmts| = 1 and
            first (nested_stmts).kind = BLOCK
        else if label.kind = IF_ELSE
          then
            |nested_stmts| = 2 and
            first (nested_stmts).kind = BLOCK and
            last (nested_stmts).kind = BLOCK
        else if label.kind = CALL
          then
            |nested_stmts| = 0)

math definition IS_INITIAL_STATEMENT (s: STATEMENT)
  : boolean is
root (s).kind = BLOCK and
children (s) = empty_string

!*/

standard_abstract_operations (Statement_Kernel);
/*! Statement_Kernel is modeled by STATEMENT
   initialization
   ensures IS_INITIAL_STATEMENT (self)
*/

procedure Add To Block (preserves Integer pos,
  consumes Nested_Statement_Type& statement)
  is abstract;
/*! requires
root (self).kind = BLOCK and
root (statement).kind /= BLOCK and
0 <= pos and pos <= |children (self)|

ensures
there exists x, y: string of STATEMENT
(|x| = pos and
children (#self) = x * y and
self = compose (root (#self), x * <#statement> * y))

procedure Remove_From_Block (  
  preserves Integer pos,
  produces Nested_Statement_Type& statement
) is_abstract;

function Integer Length_Of_Block () is_abstract;

procedure Compose_If (  
  consumes Integer& cond,
  consumes Nested_Statement_Type& block
) is_abstract;

procedure Decompose_If (  
  produces Integer& cond,
  produces Nested_Statement_Type& block
) is_abstract;
ensures
    #self = compose ((IF, cond, empty_string), <block>)
/*!*/

procedure Compose_If_Else (  
    consumes Integer& cond,  
    consumes Nested_Statement_Type& if_block,  
    consumes Nested_Statement_Type& else_block  
) is_abstract;  
/*!!
produces self  
requires  
IS_CONDITION (cond) and  
root (if_block).kind = BLOCK and  
root (else_block).kind = BLOCK  
ensures  
    self = compose ((IF_ELSE, #cond, empty_string),  
    <#if_block> * <#else_block>)
/*!*/

procedure Decompose_If_Else (  
    produces Integer& cond,  
    produces Nested_Statement_Type& if_block,  
    produces Nested_Statement_Type& else_block  
) is_abstract;  
/*!!
consumes self  
requires  
    root (self).kind = IF_ELSE  
ensures  
    #self = compose ((IF_ELSE, cond, empty_string),  
    <if_block> * <else_block>)
/*!*/

procedure Compose_While (  
    consumes Integer& cond,  
    consumes Nested_Statement_Type& block  
) is_abstract;  
/*!!
produces self  
requires  
IS_CONDITION (cond) and  
root (block).kind = BLOCK  
ensures  
    self = compose ((WHILE, #cond, empty_string),  
    <#block>)
/*!*/

procedure Decompose_While (  
    produces Integer& cond,  
    produces Nested_Statement_Type& block  
) is_abstract;  
/*!!
consumes self  
requires
```c

    root (self).kind = WHILE
    ensures
        #self = compose ((WHILE, cond, empty_string), <block>)
    */

procedure Compose_Call (
    consumes Text& inst
) is_abstract;
    /*
    produces self
    requires
        IS_IDENTIFIER (inst) and inst /= empty_string
    ensures
        root (self) = (CALL, TRUE, #inst) and
        children (self) = empty_string
    */

procedure Decompose_Call (
    produces Text& inst
) is_abstract;
    /*
    consumes self
    requires
        root (self).kind = CALL
    ensures
        #self = compose ((CALL, TRUE, inst), empty_string)
    */

function Integer Kind () is_abstract;
    /*
    ensures
        Kind = root (self).kind
    */

};

#endif // AT_STATEMENT_KERNEL
```

Last modified: Thu Jan 11 16:58:48 EST 2007
#ifndef
#define
AT_STATEMENT_PARSE 1

#include "AT/Statement/Kernel.h"
#include "AI/BL_Tokenizing_Machine/Kernel.h"

abstract_template <
    concrete_instance class Statement_Base,
    */!
    implements
    abstract_instance Statement_Kernel <
        Statement_Base
    >
    /*!
    concrete_instance class Tokenizing_Machine
    */!
    implements
    abstract_instance BL_Tokenizing_Machine_Type
    */!
>
class Statement_Parse :
    extends
    abstract_instance Statement_Base
{
public:
    /*!
    math definition CONDITION_TO_TOKEN ( 
        i: integer
    ) : string of character satisfies 
    if i = NEXT_IS_EMPTY
then CONDITION_TO_TOKEN (i) = "next-is-empty"
else if i = NEXT_IS_NOT_EMPTY
then CONDITION_TO_TOKEN (i) = "next-not-empty"
else if i = NEXT_IS_WALL
then CONDITION_TO_TOKEN (i) = "next-wall"
else if i = NEXT_IS_NOT_WALL
then CONDITION_TO_TOKEN (i) = "next-not-wall"
else if i = NEXT_IS_FRIEND
then CONDITION_TO_TOKEN (i) = "next-friend"
else if i = NEXT_IS_NOT_FRIEND
then CONDITION_TO_TOKEN (i) = "next-not-friend"
else if i = NEXT_IS_ENEMY
then CONDITION_TO_TOKEN (i) = "next-enemy"
else if i = NEXT_IS_NOT_ENEMY
then CONDITION_TO_TOKEN (i) = "next-not-enemy"
else if i = RANDOM
then CONDITION_TO_TOKEN (i) = "random"
else if i = TRUE
then CONDITION_TO_TOKEN (i) = "true"

math definition BLOCK_TO_STRING_OF_TOKENS (block: string of STATEMENT): string of string of character satisfies
if block = empty_string
then
    BLOCK_TO_STRING_OF_TOKENS (block) = empty_string
else
    there exists s: STATEMENT, rest: string of STATEMENT
    (block = <s> * rest and
    BLOCK_TO_STRING_OF_TOKENS (block) =
    STATEMENT_TO_STRING_OF_TOKENS (s) *
    BLOCK_TO_STRING_OF_TOKENS (rest))

math definition STATEMENT_TO_STRING_OF_TOKENS (s: STATEMENT): string of string of character satisfies
there exists label: STATEMENT_LABEL,
    nested_stmts: string of STATEMENT
(s = compose (label, nested_stmts) and
if label.kind = BLOCK
then
    STATEMENT_TO_STRING_OF_TOKENS (s) =
    BLOCK_TO_STRING_OF_TOKENS (nested_stmts)
else if label.kind = IF
then
    STATEMENT_TO_STRING_OF_TOKENS (s) =
    "<"IF"> * <CONDITION_TO_TOKEN (label.test)> *
    "THEN"> *
    STATEMENT_TO_STRING_OF_TOKENS (first (nested_stmts)) *
    "END"> * "IF">
else if label.kind = IF_ELSE
then
    STATEMENT_TO_STRING_OF_TOKENS (s) =
"IF" * <CONDITION_TO_TOKEN (label.test)> *
"THEN" *
STATEMENT_TO_STRING_OF_TOKENS (first (nested_stmts)) *
"ELSE" *
STATEMENT_TO_STRING_OF_TOKENS (last (nested_stmts)) *
"END" * <"IF">
else if label.kind = WHILE then
STATEMENT_TO_STRING_OF_TOKENS (s) =
"WHILE" * <CONDITION_TO_TOKEN (label.test)> *
"DO" *
STATEMENT_TO_STRING_OF_TOKENS (first (nested_stmts)) *
"END" * <"WHILE">
else if label.kind = CALL then
STATEMENT_TO_STRING_OF_TOKENS (s) =
<label.instruction>)

procedure Parse (alters Character_IStream& str,
alters Tokenizing_Machine& m,
alters Text& token_text,
alters Integer& token_kind
) is_abstract;
/*! 
produces self
requires
str.is_open = true and
token_kind = WHICH_KIND (token_text) and
m.ready_to_dispense = false
ensures
if there exists s: STATEMENT, x, y: string of character
(root (s).kind /= BLOCK and
#token_text * #m.buffer * #str.content = x * y and
STATEMENT_TO_STRING_OF_TOKENS (s) =
REMOVE_SEPARATORS (TOKENIZE_PROGRAM_TEXT (x)))
then
str.is_open = true and
str.ext_name = #str.ext_name and
there exists z: string of character
(#token_text * #m.buffer * #str.content =
z * token_text * m.buffer * str.content and
STATEMENT_TO_STRING_OF_TOKENS (self) =
REMOVE_SEPARATORS (TOKENIZE_PROGRAM_TEXT (z)) and
root (self).kind /= BLOCK and
token_kind = WHICH_KIND (token_text) and
m.ready_to_dispense = false)
/*! 
procedure Parse_Block (}
alters Character_IStream& str,
alters Tokenizing_Machine& m,
alters Text& token_text,
alters Integer& token_kind
) is_abstract;
/*! produces self
 requires
   str.is_open = true and
ten_kind = WHICH_KIND (token_text) and
m.ready_to_dispense = false
ensures
   if there exists x, y: string of character,
    s: string of STATEMENT
       (#token_text * #m.buffer * #str.content = x * y and
        BLOCK_TO_STRING_OF_TOKENS (s) =
        REMOVE_SEPARATORS (TOKENIZE_PROGRAM_TEXT (x)))
    then
      str.is_open = true and
      str.ext_name = #str.ext_name and
      there exists z: string of character
         (#token_text * #m.buffer * #str.content =
          z * token_text * #m.buffer * #str.content and
          BLOCK_TO_STRING_OF_TOKENS (children (self)) =
          REMOVE_SEPARATORS (TOKENIZE_PROGRAM_TEXT (z)) and
          root (self).kind = BLOCK and
          token_text is not in {"IF", "WHILE"} and
          not IS_IDENTIFIER (token_text) and
          not IS_WHITE_SPACE (token_text) and
          not IS_COMMENT (token_text) and
          token_kind = WHICH_KIND (token_text) and
          m.ready_to_dispense = false)
   */
};
#endif // AT_STATEMENT_PARSE

Last modified: Mon Apr 02 10:04:51 EDT 2007
Abstract Template : Statement_Pretty_Print

global context

interface

abstract_template <
  concrete_instance class Statement_Base
  
  implements
  
  abstract_instance Statement_Kernel <
    Statement_Base
  >
  
  !*/
>

class Statement_Pretty_Print :
  extends
    abstract_instance Statement_Base
{
  public:
    
    math definition MAKE_A_STRING ( i: integer,
        ch: character
    ) : string of character satisfies
    if i <= 0
    then
      MAKE_A_STRING (i, ch) = empty_string
    else
      MAKE_A_STRING (i, ch) = <ch> * MAKE_A_STRING (i-1, ch)
    
    math definition DISPLAY_CONDITION ( i: integer

}): string of character satisfies
if i = NEXT_IS_EMPTY
then DISPLAY_CONDITION (i) = "next-is-empty"
else if i = NEXT_IS_NOT_EMPTY
then DISPLAY_CONDITION (i) = "next-is-not-empty"
else if i = NEXT_IS_WALL
then DISPLAY_CONDITION (i) = "next-is-wall"
else if i = NEXT_IS_NOT_WALL
then DISPLAY_CONDITION (i) = "next-is-not-wall"
else if i = NEXT_IS_FRIEND
then DISPLAY_CONDITION (i) = "next-is-friend"
else if i = NEXT_IS_NOT_FRIEND
then DISPLAY_CONDITION (i) = "next-is-not-friend"
else if i = NEXT_IS_ENEMY
then DISPLAY_CONDITION (i) = "next-is-enemy"
else if i = NEXT_IS_NOT_ENEMY
then DISPLAY_CONDITION (i) = "next-is-not-enemy"
else if i = RANDOM
then DISPLAY_CONDITION (i) = "random"
else if i = TRUE
then DISPLAY_CONDITION (i) = "true"

math definition DISPLAY_BLOCK (s: string of STATEMENT
i: integer
): string of character satisfies
if s = empty_string
then DISPLAY_BLOCK (s, i) = empty_string
else
there exists stmt: STATEMENT,
rest: string of STATEMENT
(s = <stmt> * rest and
DISPLAY_BLOCK (s, i) =
PRETTY_DISPLAY (stmt, i) * DISPLAY_BLOCK (rest, i))

math definition PRETTY_DISPLAY (s: STATEMENT
i: integer
): string of character satisfies
there exists label: STATEMENT_LABEL,
nested_stmts: string of STATEMENT
(s = compose (label, nested_stmts) and
if label.kind = BLOCK
then
PRETTY_DISPLAY (s, i) = DISPLAY_BLOCK (nested_stmts, i)
else if label.kind = IF
then
PRETTY_DISPLAY (s, i) =
MAKE_A_STRING (i, ' ') "IF " * DISPLAY_CONDITION (label.test) * " THEN\n" *
PRETTY_DISPLAY (first (nested_stmts), i+4) *
MAKE_A_STRING (i, ' ') "END IF\n"
else if label.kind = IF_ELSE
then
PRETTY_DISPLAY (s, i) =
MAKE_A_STRING (i, ' ') * "IF " *
DISPLAY_CONDITION (label.test) * " THEN\n" *
PRETTY_DISPLAY (first (nested_stmts), i+4) *
MAKE_A_STRING (i, ' ') * "ELSE\n" *
PRETTY_DISPLAY (last (nested_stmts), i+4) *
MAKE_A_STRING (i, ' ') * "END IF\n"
else if label.kind = WHILE then
PRETTY_DISPLAY (s, i) =
MAKE_A_STRING (i, ' ') * "WHILE " *
DISPLAY_CONDITION (label.test) * " DO\n" *
PRETTY_DISPLAY (first (nested_stmts), i+4) *
MAKE_A_STRING (i, ' ') * "END WHILE\n"
else if label.kind = CALL then
PRETTY_DISPLAY (s, i) =
MAKE_A_STRING (i, ' ') * label.instruction * "\n"
end if