Don Rodolfo P. Abril
Jerico L. Silapan

Asin LANGUAGE DESIGN &SPECIFICATIONS REPORT

A. *Introduction*

I. Language Name

As one would see in the title header above, our language's name is Asin.

We named it Asin as it would remind us of the salt in the proverbial sweat and tears shed throughout CS 150, and especially throughout the creation of this language.

II. Programming Paradigm

Asin is an imperative language. We opted for an imperative paradigm since imperative languages are the ones we have the most experience with (C, C++, Python, Java).

III. Language Inspiration

Asin is inspired by Python and C, taking familiar elements from both (though mostly from Python).

For more information while reading this report, you may find it convenient to refer to the Asin User Manual.

B. *Grammar Definition*

First off, take some time to familiarize yourself with the lexeme/token/terminal mapping below (found in `asinlex.py`), so that you may better understand the grammar (found in `asinyacc.py`).

LEXEME/TOKEN/TERMINAL	VALUE/REGULAR EXPRESSION
Basic units (excluding Boolean values)	
ID	r'[_a-zA-Z][_a-zA-Z0-9]*'
INTEGER	r'\d+'
FLOAT	r'\d*\.\d*'
STRING	r'"([\\.]|[^"\\])*"'
NEWLINE (only used for lexer)	r'\n+'
Arithmetic operators	
PLUS	+
MINUS	-
MUL	*
DIV	/
FDIV	//
EXP	**
MOD	%

Comparison operators	
EQ	==
NEQ	!=
GT	>
GTE	>=
LT	<
LTE	<=
Grouping symbols	
LPAREN	(
RPAREN	)
LSQUARE	[
RSQUARE	]
LCURLY	{
RCURLY	}
Assignment symbols	
EQUALS	=
PLUSEQUALS	+=
MINUSEQUALS	-=
MULEQUALS	*=
DIVEQUALS	/=
FDIVEQUALS	//=
EXPEQUALS	**=
MODEQUALS	%=
Separators	
COMMA	,
COLON	:
SEMICOLON	;
Conditionals	
IF	kapag
BUT	ngunit
ELSE	kundiman
Loops	
WHILE	hanggat
FOR	bawat
IN	sa
EXIT	lumisan
Printing	
PRINT	ilimbag
Boolean values	
TRUE	Totoo
FALSE	Huwad
Logical truth value operators	
AND	at
OR	o
NOT	hindi

The entire grammar (let's call this the master grammar for the sake of convenience) of Asin can be seen as follows. Words in lowercase represent non-terminals, while words in uppercase are the terminals.

Recursive rule for representing all lines of code

```
statementblock : statementblock statement
               | statement
```

Rule for statements (i.e, constructs/structures essential to program flow)

```
statement : assign_statement
          | comp_assign_statement
          | if_statement
          | loop_statement
          | print_statement
          | function_call_statement
          | exit_statement
```

Rule for assignment statements, as in, for assigning data to variables

```
assign_statement : identifier EQUALS expression SMCOLON
```

Rule for compound assignment statements, which combine simple math and assignments into singular statements

```
comp_assign_statement : identifier PLUSEQUALS expression SMCOLON
                      | identifier MINUSEQUALS expression SMCOLON
                      | identifier MULEQUALS expression SMCOLON
                      | identifier DIVEQUALS expression SMCOLON
                      | identifier FDIVEQUALS expression SMCOLON
                      | identifier EXPEQUALS expression SMCOLON
                      | identifier MODEQUALS expression SMCOLON
```

Rule for conditional statements, i.e. decision-making

```
if_statement : IF LPAREN expression RPAREN LCURLY statementblock RCURLY
             | IF LPAREN expression RPAREN LCURLY statementblock RCURLY
               ELSE LCURLY statementblock RCURLY
             | IF LPAREN expression RPAREN LCURLY statementblock RCURLY BUT
               if_statement
```

General rule for loop constructs, which chain to the more specific while- and for-loop rules

```
loop_statement : while_statement
               | for_statement
```

Rule for while-loops; for iterative program instructions

```
while_statement : WHILE LPAREN expression RPAREN LCURLY statementblock
                  RCURLY
```

For-loops' rule; for iterative program instructions

```
for_statement : IN FOR identifier IN LSQUARE expression COLON expression
                RSQUARE LCURLY statementblock RCURLY
```

Rule for print statements; for printing out values to the CLI

```
print_statement : PRINT LPAREN commasepexpr RPAREN SMCOLON
```

Rule for function call statements; for performing functions that are designed as statements

```
function_call_statement : function_call SMCOLON
```

Rule for function calls

```
function_call : identifier LPAREN commasepexpr RPAREN
              | identifier LPAREN RPAREN
```

Rule for exit statements, that allow the breaking of loops

```
exit_statement : EXIT SMCOLON
```

Rule for identifiers (names), which are attached to constants, variables and functions

```
identifier : ID
```

Rule for primitives, the most basic elements and data types of the language (integer, float, string, Boolean)

```
primitive : INTEGER
          | FLOAT
          | STRING
          | TRUE
          | FALSE
```

Rule for Boolean expressions, which evaluate to either Totoo (True) or Huwad (False)

```
expression : expression EQ expression
           | expression NEQ expression
           | expression GT expression
           | expression GTE expression
           | expression LT expression
           | expression LTE expression
           | expression AND expression
           | expression OR expression
```

Rule for arithmetic expressions, which evaluate to numbers

```
expression : expression PLUS expression
           | expression MINUS expression
           | expression MUL expression
           | expression DIV expression
           | expression FDIV expression
           | expression EXP expression
           | expression MOD expression
```

Rule for unary operations; mathematical and logical negation

```
expression : MINUS expression
           | NOT expression
```

Rule for grouped expressions, which allow better control for following math's PEMDAS rule

```
expression : LPAREN expression RPAREN
```

Rule for creating lists, the 5th data type in Asin

```
expression : LSQUARE commasepexpr RSQUARE
           | LSQUARE RSQUARE
```

Rule for accessing list elements

```
expression : identifier LSQUARE expression RSQUARE
```

Rule for expanding expressions into base values and function return values

```
expression : primitive
           | identifier
           | function_call
```

Rule for comma-separated elements of code, such as function parameters and list contents

```
commasepexpr : commasepexpr COMMA expression
             | expression
             | ε
```

Note that while the grammar rules for the expansion of `expression` might look redundant, they are defined through the yacc file for different and appropriate placements in the syntax of a file that would be parsed. The same redundancy also produced ambiguity and caused many shift/reduce errors.

However, these S/R errors were raised by yacc for tokens in the LALR states where the expected action otherwise (i.e if the grammar were unambiguous) would be to shift, and since Python's ply/yacc favors a shift action in the event of an S/R error, then the grammar works almost just as well as if it were truly unambiguous; the only exceptions are comparison operations (i.e ==, !=, >, <, etc.) which would err if the operand on both sides are non-primitive expressions that aren't grouped.

Identifiers/constants in Asin are defined by the following regular expression, as seen in the token-regex mapping above:

```
r'[a-zA-Z][_a-zA-Z0-9]*'
```

They are set and stored as keys in a dictionary (hash table) that point to their respective values. Such values may take on the types of data listed below.

Asin **data types** include integer, float, string, Boolean, and list, the same as in Python's. All of these are primitives (primitive). Being built on Python, we deemed it ideal to adopt its type inference (implicit typing) feature. See the following definitions for how these types are tokenized.

```
def t_INTEGER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_FLOAT(t):
    r'\d*\.\d*'
    t.value = float(t.value)
    return t

def t_STRING(t):
    r'"(?:\\"|.)*?"'
    t.value = t.value.lstrip('"')
    t.value = t.value.rstrip('"')
    return t

def t_TRUE(t):
    r'Totoo'
    t.value = True
    return t

def t_FALSE(t):
    r'Huwad'
    t.value = False
    return t
```

Lists, on the other hand, are not tokenized but are defined through the grammar rule for lists. A list's elements may exhibit any combination of data types above, along with identifiers.

```
expression : LSQUARE commasepexpr RSQUARE
           | LSQUARE RSQUARE
```

`commasepexpr` refers to the contents of the list.

Assignment statements take the following form.

```
statement : assign_statement
assign_statement : identifier EQUALS expression SMCOLON
```

Compound assignment statements can also be generated with the grammar.

```
statement : comp_assign_statement
comp_assign_statement : identifier PLUSEQUALS expression SMCOLON
                      | identifier MINUSEQUALS expression SMCOLON
                      | identifier MULEQUALS expression SMCOLON
                      | identifier DIVEQUALS expression SMCOLON
                      | identifier FDIVEQUALS expression SMCOLON
                      | identifier EXPEQUALS expression SMCOLON
                      | identifier MODEQUALS expression SMCOLON
```

The pertinent value in any assignment statement is an **expression**. There are several rules defined for expressions. For explanations, refer to the master grammar in page 4.

```
expression : expression EQ expression
           | expression NEQ expression
           | expression GT expression
           | expression GTE expression
           | expression LT expression
           | expression LTE expression
           | expression AND expression
           | expression OR expression

expression : expression PLUS expression
           | expression MINUS expression
           | expression MUL expression
           | expression DIV expression
           | expression FDIV expression
           | expression EXP expression
           | expression MOD expression

expression : MINUS expression
           | NOT expression

expression : LPAREN expression RPAREN

expression : LSQUARE commasepexpr RSQUARE
           | LSQUARE RSQUARE

expression : identifier LSQUARE expression RSQUARE

expression : primitive
           | identifier
           | function_call
```

The grammar rules of (if-else) **conditional statements** are as follows:

```
statement : if_statement
if_statement : IF LPAREN expression RPAREN LCURLY statementblock RCURLY
             | IF LPAREN expression RPAREN LCURLY statementblock RCURLY
               ELSE LCURLY statementblock RCURLY
             | IF LPAREN expression RPAREN LCURLY statementblock RCURLY BUT
               if_statement
```

`expression` can be anything that would evaluate to a Boolean value, possibly through recursions of the same rule. `statementblock` refers to instructions inside the clauses of conditional statements.

For generating **iterative statements**, refer to the grammar rules below.

```
statement : loop_statement
loop_statement : while_statement
               | for
while_statement : WHILE LPAREN expression RPAREN LCURLY statementblock
                  RCURLY
for_statement : IN FOR identifier IN LSQUARE expression COLON expression
                RSQUARE LCURLY statementblock RCURLY
```

For while-loops, `expression` can be anything that would evaluate to a Boolean value, possibly through recursions of the same rule. `statementblock expression` holds true.

For for-loops, the `expression`(s) can be anything that can be reduced to an integer value.

Function calls/statements/expressions take the following form.

```
function_call_statement : function_call SMCOLON

expression : function_call

function_call : identifier LPAREN commasepexpr RPAREN
              | identifier LPAREN RPAREN
```

`identifier` refers to the function name, while `commasepexpr` pertains to the parameters to be passed to the function.

For the usage of functions, please refer to the user manual where they are more fleshed out. Asin's **input and output functions** are also specified there, taking on the same form as the grammar rule above.
