CS-315

FALL 2022

TERM PROJECT PART-B

Programming Language: Strada

Instructor: Halil Altay Güvenir

Section:01

Group: 46

Team Members:

Izaan Aamir 22001488

Tuna Okçu 22002940

Deniz Hayri Özay 21803632

1. Tokens Defined with Regular Expressions:

The tokens mentioned below are defined as regular expressions. The first three tokens are basic tokens that are used when defining Strings, Chars, Floats, and Integers. We have also added Delimiters as they will assist in writing and defining characters and Strings.

DIGIT ::= “0”|”1”|”2”|…|”9”
ALPHABETS ::= “A”|”B”|”C”|…|”Z”|”a”|”b”|”c”|….|”y”|”z”
SIGN ::= +|-
DELIMETER ::= '!' | '"' | '#' | '$' | '%' | '&' | ''' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | '<' | '=' | '>' | '?' | '@' | '[' | '\' | ']' | '^' | '_' | '|' | '}' | '{' | '~'

These tokens are non-trivial and are specific to our language, Strada. In our language, comments are defined as anything between /* and */ giving us a single line comment. Furthermore, we also have identifiers defined as any literal that might be used in order to identify any variable, method or function.

COMMENT ::= /* DIGIT | ALPHABET | DELIMITER*/
IDENTIFIER ::= DIGIT | ALPHABET | DELIMITER
AND ::= “&&”
OR ::= “||”

The tokens defined below such as Node and Node_Methods are specific to our programming language to support the use of the IoT devices. These nodes can be declared and then the relevant methods can be used accordingly. This adds to the readability and writability of the programming language as it aids in the programmer with working quicker with IoT devices.

- **flipSwitch(n) toggles the nth switch(switches range from switch 1 to 10)**
- **isOn(n) returns boolean value true if the nth switch is on and false if its off**
- **time() returns a long value representing the current time in seconds elapsed(“number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds”)**

- **connect(str) connects the Node it's called on to the URL specified in str**
- **getConnectionNames() returns an array of str values that represent the connections**
- **removeConnection(str) disconnects the Node from the URL specified in str**
- **send(str, val) sends a value of type int to a connection specified**
- **receive() returns incoming value of type int**
- **getSensor(str) returns the latest reading of the sensor in float form**
- **getSensorNames() returns a string array consisting of the names of the sensors attached to the node**

NODE ::= "Node"
NODE_METHODS ::= "flipSwitch" | "isOn" | "time" | "connect" | "send" | "receive" | "addSensor" | "setSensor" | "getSensor" | "getSensorNames" | "removeSensor"

These tokens are specific to declaring data values. Our language supports Integers, Long, Characters, Boolean and Strings. We chose these different from identifiers so that it is more readable to the programmer.

INT_VALUE ::= SIGN? DIGIT
LONG_VALUE ::= SIGN? DIGIT
FLOAT_VALUE ::= SIGN? DIGIT* . DIGIT+
CHAR_VALUE ::= DELIMITER | DIGITS | ALPHABETS
STRING_VALUE ::= CHAR_VALUE
BOOL_VALUE ::= true | false

For readability, we have added return and break statements so that it is easier for the programmer to use the language and have a modular approach. This makes it more readable and writable.

RETURN ::= "return"
BREAK ::= "break"
WHILE ::= "while"
FOR ::= "for"

All the following tokens are going to be used for any arithmetic computation or assignment operation. Furthermore, they can also be used to evaluate any relational expressions and assign values to identifiers. These also aid in writibility(easier to write a += b than a = a + b).

PLUS_ASSIGN ::= “+=”
MINUS_ASSIGN ::= “-=”
LESS_THAN ::= “<”
GREATER_THAN ::= “>”
EQUALITY ::= “==”
INEQUALITY ::= “!=”
LESS_EQ ::= “<=”
GREATER_EQ ::= “>=”

2. Strada Definition

1) Program:

<program> ::= <sub-program>
<sub-program> ::= <empty> | <stmt-list>
<stmt-list> ::= <stmt-list> <stmt> | <stmt>
<stmt> ::= <simple-stmt> | <compound-stmt> | NEWLINE | COMMENT
<type-specifier> ::= int | float | char | bool | string | long

<identifier-list>::= IDENTIFIER | IDENTIFIER, <identifier-list>
<empty> ::=

2) Simple Statements

<simple-stmt> ::= <declaration-stmt> | <assignment-stmt> | <return-stmt> | <break-stmt> | <expression>
<declaration-stmt> ::= <type-specifier> <identifier-list>;| <type-specifier><assignment-stmt> | <array-declaration> | <node-declaration>
<array-declaration> ::= <type-specifier> <array-index>;

<assignment-stmt> ::= IDENTIFIER <assignment-operator><expression> ; | <array-assignment>
<array-assignment> ::= <array-index> <assignment-operator> <expression> ;

<assignment-operator> ::= = | PLUS_ASSIGN | MINUS_ASSIGN

<return-stmt> ::= RETURN <expression> ; | RETURN ;
<break-stmt> ::= BREAK ;

3) Compound Statements

<compound-stmt> ::= <function-definition-stmt> | <loop-stmt> |
<conditional-stmt>

<function-definition-stmt> ::= <type-specifier> IDENTIFIER (<param-list>) {
<sub-program> }

<param-list> ::= <empty> | <full-param-list>
<full-param-list> ::= <parameter> | <parameter> , <param-list>
<parameter> ::= <type-specifier> IDENTIFIER

<loop-stmt> ::= <while-loop> | <for-loop>
<while-loop> ::= WHILE (<bool-expression>) { <stmt-list> }
<for-loop> ::= FOR (<type-specifier> <assignment-stmt> <bool-expression> ;
<simple-stmt>) { <stmt-list> }
<conditional-stmt> ::= <if-stmt> | <if-else-stmt>
<if-stmt> ::= IF (<bool-expression>) { <sub-program> }
<if-else-stmt> ::= IF (<bool-expression>) {<sub-program> } ELSE
{ <sub-program> }

4) Expressions

<expression> ::= <bool-expression>

<simple-expression> ::= <constant-expression> | <array-index> |
<func-call-expression> | <method-call-expression> | IDENTIFIER

<constant-expression>::= INT_VALUE | FLOAT_VALUE | ‘CHAR_VALUE’ |
“STRING_VALUE” | BOOL_VALUE | LONG_VALUE

<array-index> ::= IDENTIFIER [<expression>]
<func-call-expression> ::= IDENTIFIER (<argument-list>)

<arithmetic-expression> ::= <arithmetic-expression> + <term>
| <arithmetic-expression> - <term>
| <term>
<term> ::= <term> / <simple-expression>
| <term> * <simple-expression>
| <simple-expression>

<bool-expression> ::= <bool-expression> OR <bool-term> | <bool-term>
<bool-term>::= <bool-term> AND <relational-term> | <relational-term>
<relational-term>::= <bool-factor> <relational-operators> <bool-factor> | <bool-factor>
<relational_operators> ::= LESS_THAN | GREATER_THAN | EQUALITY | INEQUALITY | LESS_EQ | GREATER_EQ
<bool-factor> ::= arithmetic_expression | NOT arithmetic_expression

<argument-list> ::= <empty> | <full-argument-list>
<full-argument-list> ::= <expression> | <expression> , <argument-list>

<relational-operators>::= LESS_THAN | GREATER_THAN | EQUALITY |
INEQUALITY | LESS_EQ | GREATER_EQ
<bool-operators> ::= AND | OR

5) **Nodes**

<node-declaration> ::= NODE IDENTIFIER ;
<method-call-expression> ::= IDENTIFIER . NODE_METHODS (<argument-list>)

3. Strada Definition Explanations

Program

<program> : Our start symbol

<sub-program> : Any subprogram. Either a list of statements or empty.

<stmt-list> : One or more statements.

<stmt>: Simple or compound statement(based on how many lines the statement is going to span); new line or comment.

<type-specifier>: Type specifier for variable declarations and function signature. Node is excluded because this class has its own unique grammars(Node x = 10; is not a legal statement for example).

<identifier-list>: For quicker variable declaration of multiple types(int x, y, z;).

<empty> ::= Nothing.

Simple Statements

<simple-stmt>: A statement that spans one line: i.e. it is concluded with a semicolon(SC).

<declaration-stmt>: Declaration statement for one or more variables. A single variable can immediately be initialized. Multiple variables declared at once cannot be initialized in the declaration. An array can be declared but not initialized in the declaration.

<array-declaration>: Array declaration(ex: int arr[10]).

<assignment-stmt>: Variable assignment(ex: x = 10+9;) or array assignment(ex: arr[9] = f(10);). The right hand side of an assignment statement is any expression, allowing the assignment of compound expressions(ex: max(3,9) + arr[4] - 1 / 9).

<array-assignment>: Assignment to the specified index of an array. Index ranges 0 to length-1 where length is the length of the array.

<assignment-operator>: A list of assignment operators(=, +=, -=).

<return-stmt>: Statement for returning from a function, with or without a value.

<break-stmt>: Statement for breaking from a loop.

Compound Statements

<compound-stmt> : Statement that can include multiple simple statements. They are concluded with a }(RBRACE).

<function-definition-stmt>: For defining a function. A return statement is not necessary at the end and a function with no return statement will simply return nothing. There is no void type so any function(for example, a function defined as int f() { }) can return nothing, regardless of its type.

<param-list>: List of parameters used in the function definition. Either empty or full-param-list.

<full-param-list>: List of parameters used in the function definition.

<parameter>: Part of <param-list> used in the function definition. Includes type and parameter name.

<loop-stmt>: For and While loops.

<while-loop>: While loop.

<for-loop>: For loop. Unusual in that the final statement in the for loop also requires a semi colon at the end. Example: for(int i = 0; i < 10; i++;){ … }

<conditional-stmt>: If and if-else. Else if structure is present in the language because of time constraints.

<if-stmt>: If statement.

<if-else-stmt>: If-else statement.

Expressions

<expression>: Any expression. It evaluates to a <constant-expression>(or a literal). A simple expression is a <bool-expression> which in its simplest form reduces to an <arithmetic-expression>.

<constant-expression>: These refer to literals. 5 is for example an INT_VALUE(a constant expression).

<array-index>: Either the resulting value of an array indexing operation(for example arr[0]) to be used in an expression(i.e. it appears in the right-hand-side) or where an assignment(for example arr[3] = 5;) will be made(i.e. it appears in the left-hand-side).

<func-call-expression>: Function call. The return value is a <constant-expression> or nothing.

<compound-expression>: Arithmetic expression or boolean expression. This separation was made to enforce associativity and operator precedence rules.

<arithmetic-expression>: Created to enforce associativity and operator precedence rules. It can either immediately evaluate to a <constant-expression>(INT_VALUE, FLOAT_VALUE, CHAR_VALUE, etc), an <array-index>(ex: arr[index]), a <func-call-expression>(ex: f(10)), an IDENTIFIER i.e. it is the immediate result of an expression; or it can evaluate to the result of an arithmetic operation involving these terms.

<term>: Created to enforce associativity and operator precedence rules. It is recursive and its base case is <simple-expression>, which does not include any arithmetic expression(so that the order is kept).

<bool-expression>: Either bool-term or bool-expression OR bool-term. These OR's can be chained forever.

<bool-term>: Either relational-term or relational-term AND bool-term. These AND's can be chained forever. AND appears one rule deeper than OR, giving it higher precedence.

<relational-term>: Either bool-factor or bool-factor relational-operators bool-factor. This gives relational operators higher precedence than AND and OR. There's a limitation in writability in that these relational operations cannot be chained: for example, a < b == c >= d is not accepted in our language.

<bool-factor>: Either arithmetic-expression or NOT arithmetic-expression. This gives NOT the highest precedence in boolean operators.

<argument-list>: Contains the list of arguments that can be sent to the functions. Either full-argument-list or empty.

<full-argument-list>: The argument list of a function can be an expression or can be more than one expression. Therefore the writibility of Strada increases dramatically.

<relational-operators>: A list of all relational operators(operators that can be applied on two simple expressions).

<bool-operators>: Bool operators are defined as binary boolean operators. The unary !(NOT) was not categorized under this variable for this reason, and for the readability of the document itself we did not divide bool-operators into binary-bool-operators and unary-bool-operators, one of which would only include a single operator.

Nodes

<node-declaration>: For initializing an object of type Node(the only class in the language) .

<method-call-expression>: Node is the only thing in our language resembling a class, therefore there are only node methods to call. There is a number of these methods available, and they are described earlier in the report.

4. Motivations and Constraints behind Strada

We have tried to achieve a controlled and balanced set of language definitions using Strada. It is lightweight, easy to use because of its readability and familiar to other imperative languages so that it doesn't have a huge learning curve for newer programmers. The language gives tons of writibility extensions as it allows the programmers to define nested loops, define their own comments using whatever characters they deem necessary and overall convey the message easily. We have also felt the need to establish Clean Code so we used curly braces whenever the user declares a conditional statement or a loop and used semi-colons to ensure that the readability of the language is maintained. Additionally, we have also configured methods for the programmer to use readily and declared it in the BNF as well as lexical analyzer. These methods will be recognized readily and quickly. This

facilitates the programmer in configuring nodes and invoking methods. It should also be noted that our parser encounters no conflicts.