Mahathi Vempati - 20161003
Jayitha. C - 20171401

Tinki* Language Manual

This document describes the syntax and semantics of the language designed as part of the course Compilers(CSE419). To design out language we researched grammars of a multitude of programming languages such a Python, Decaf and C. This language has loosely been based on Decaf and C with a few modifications

Syntax

In this section we'll proceed to formally define the macro and micro syntax of our language. We'll assume a Top Down approach where we first describe the macro syntax and then describe the micro syntax using context free grammar and regexes.

Note: The below specification is not a CFG.

Metasyntax

This section describes the grouping and quantitative notations used to describe our grammar.

```
<> -> Non terminal

[x] -> Zero or one occurrence of x

x* -> Zero or more occurrences of x

x+ -> Comma separated list of one or more occurrernces of x

{ } -> Grouping

| -> Alternatives (or)
```

Macrosyntax

```
<program> -> class Program '{' <var_decl>* <method_decl>* '}'

<var_decl> -> <type> {<id>
                | <id>'['<int_literal>']'
                | <id>'['<int_literal>']['<int_literal>']'}

<method_decl> -> {<type> | <type>'['<int_literal>']' |
<type>'['<int_literal>']['<int_literal>']' | void} <id>'('[{<type> <id>
{'['<int_literal>']' | '['<int_literal>']['<int_literal>']'}+]')' <block>

<block> -> '{' <var_decl>* <statement>* '}'
```

```
<statement> -> <location> '=' <expr>
              | <method_call>
              | if(<expr>) <block> [else <block>]
              | <expr> ? <statement> : <statement>
              | while(<expr>) <block>
              | for([<id>=<expr>]; [<expr>]; [<expr>]) <block>
              | return [<expr>]
              | break
              | continue
              | <block>

<method_call> -> callout(<string_literal> [,<callout_arg>+])

<location> -> <id>
              | <id>'['<expr>']'
              | <id>'['<expr>']['<expr>']'

<expr> -> <location>
        | <method_call>
        | <literal>
        | <expr> <bin_op> <expr>
        | - <expr>
        | ! <expr>
        | (<expr>)

<callout_arg> -> <expr> | <string_literal>

<bin_op> -> <arith_op> | <rel_op> | <eq_op> | <cond_op>

<literal> -> <int_literal> | <char_literal> | <bool_literal>
```

Microsyntax

RegEx	Token
".*"	<string_literal>
'.' | EOF	<char_literal>
true|false	<bool_literal>
[0-9]+	<int_literal>
[a-zA-Z][a-zA-Z0-9]*	<id>
== | !=	<eq_op>
<= | >= | < | >	<rel_op>
[\+ \- * / %]	<arith_op>
\\\| \\\| |&&	<cond_op>

RegEx	Token
+= | -= | =	<assign_op>
int | uint | bool | char	<type>

Example Program

This is an example program that checks whether an input number is prime.

```
1  class Program
2  {
3    void main()
4    {
5      int N;
6      int i;
7      callout("print", "Enter N: ");
8      N = callout("read", "int");
9      if(N == 1)
10      {
11        callout("print", "1 is neither prime nor composite");
12        return;
13      }
14      callout("print", "int", N);
15      if(N == 2 || N == 3)
16      {
17        callout("print", " is a prime number");
18        return;
19      }
20
21      for(i = 2; i <= N / 2; i = i + 1)
22      {
23        if(N % i == 0)
24        {
25          callout("print", " is NOT a prime number");
26          return;
27        }
28      }
29      callout("print", "is a prime number");
30      return;
31    }
32  }
```

Line 1: Every program must be enclosed in a class Program block.

Line 3: Every program that needs to run standalone should contain a main function and execution automatically starts from the main function.

Line 5: In every block, the variable declarations occur before method declarations. This makes it easier for semantic checks - as this automatically ensures no variable is used before it is declared.

Line 7: I/O is implemented as "read" and "print" arguments to the callout method. These functions will be defined in the lib file.

Line 30: A return statement in the main function exits the program.

Note: `for`, `if-else` and `while` have the same semantics as that of C language.

Semantic Checks

It is possible for statements to be produced by the given grammar, and therefore be completely syntactically correct, but still be useless, meaningless or wrong. This is the semantic aspect of the program.

While it is not possible to eliminate all symantic errors at compile time, a few can be taken care of:

Type Mismatch

We check every assignment statement to ensure that the assigned data types to identifiers or literals on both sides match.

Undeclared Variable

Ensure that every identifier in use has been assigned a type.

Redeclaring Variables

Redeclaring variables, especially if they already contain data is not allowed.

Accessing a variable out of scope

The compiler would need to keep tab of variables declared in given environments.

Method Call Arguments and Method Parameter Mismatch

Compiler checks every method call against method definition.

Modifications from Decaf

- Ability to handle strings and characters
- Arguments to methods and method return types can be arrays
- 'While' control flow function has been added
- Ternary operator
- Restricted function calls - can only use callout to call functions with arguments. This makes it easier to parse.