University of Pisa - Scuola Superiore Sant'Anna

Department of Computer Science

TinyJs Programming Language

Report

Advanced Programming

Prof. Gian Luigi Ferrari:

Prof. Antonio Cisternino

Team members:

1. Alemu Tadesse Metaferia
2. Kebede Sahile Akalewold
3. Temesgen Taye Lamesa

MSc in Computer Science and Networking

Academic Year 2014/2015

Table of Contents

1. INTRODUCTION

TinyJs is developed as the final project for the 1st year's Advanced Programming course of the MSc in Computer Science and Networking department. The objective of this report is to give an overview of the TinJs Programming Language implementation.

TinyJs is a JavaScript programming language which contains a small set of data types. It has the three primitive types Boolean, Number, and String and the special valued Undefined. It has a standard operators, control flow statements, conditional statements and function.

The developing team organized the designing of each structural level of the project which includes the Tokenizer/Lexer, Parser and Interpreter of the language.

2. LANGUAGE DESIGN

The designing phase has started by defining the grammar of the language which is a set of rules describing the valid expression and statements of the language.

Language basics

Variables: Variables have no type attached, and any value can be stored in any variable. A variable declared without a value will have UNDEFINED value.

Operators: TinyJS has arithmetic, logical and relational operators which are similar to C programming language.

- ✓ Arithmetic operators: - (+, -, *, /) can be either unary or binary operator
- ✓ Relational operators :- (==, !=, >, >=, <, <=)
- ✓ Logical operators:- (&&, ||)

Expressions: - it is made up of variables, operators and method invocation, which are made according to the syntax of the language. An expression evaluate to a single value.

Statements: Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution. Supported statements include: For, while, If…else, Switch, break and continue. A pair of curly brackets { } and an enclosed sequence of statements constitute a block, which can be used wherever a statement can be used.

Functions: Functions look like C functions but they are declared with **function** keyword instead of type. In the definition we can have sequence of statements, possibly empty parameter list and name. A function may define its local variables via **var**.

eval (expression) Function: The eval() function evaluate a TinyJs code represented as a string.

3. GRAMMAR

The semantics of BNF is used to express the grammar of the TinyJS language and the ambiguity, left recursion and left factor issues have been considered in writing the grammar.

Program: MixStatements

MixStatements : Statement MixStatements | Declaration MixStatements | "ε"

Declaration: "var" "Identifier" VarList ";" | Function

VarList: MoreVar | "=" Exp MoreVar | "ε"

MoreVar: "," "Identifier" VarList | "ε"

Function: "function" "Identifier" "(" ParameterList ")" Block

Block: "{" MixStatements "}"

ParameterList: "Identifier" MoreParams

MoreParams: "," ParameterList | "ε"

Statement:

|"if" Condition Block ElseStmt

|"while" Condition Block

|"for" "(" "Identifier" "=" Exp ";" Exp ";" "Identifier" StmtIde ")" Block

|SwitchStmt

|"break" ";"

|"continue" ";"

|"return" Exp";"

|"println" "(" Exp ")" ";"

|"Identifier" "=" Exp;

|EvalExp ";"

|"Identifier" StmtIde ";"

|"NUMBER" StmtIde ";"

ElseStmt: "else" Block |"ε"

Condition: "(" Exp ")"

SwitchStmt: "switch" "(" "Identifier" ")" "{" CaseBlock "}"

CaseBlock: CaseClause CaseClauses | DefaultCase

CaseClauses: CaseBlock | "ε"

CaseClause: "case" Const ":" MixStatements

DefaultCase: "default" ":" MixStatements

EvalExp = "eval" "(" EvalParam ")"

EvalParam = STRING | "Identifier"

Exp: AndExp MoreAndExps

MoreAndExps: "||" AndExp | "ε"

AndExp: UnaryRelExp MoreUnaryRelExps

MoreUnaryRelExps: "&&" UnaryRelExp | "ε"

UnaryRelExp: "!" UnaryRelExp | RelExp

RelExp: SumExp MoreSumExps

MoreSumExps: RelOp SumExp | "ε"

RelOp: "<=" | "<" | ">" | ">=" | "==" | "!="

StmtIde: Exp | "+=" Exp | "-=" Exp | "++" | "--"

SumExp: Term MoreTerms

MoreTerms: "+" Term | "-" Term | "ε"

Term: UnaryExp MoreUnaryExps

MoreUnaryExps: "*" UnaryExp | "/" UnaryExp | "ε"

UnaryExp: "-" UnaryExp | Factor

Factor:"(" Exp ")" | Const|CALL_IDENT

CALL_IDENT:"Identifier"|Call

Call: "Identifier" "(" Params ")"

Params: ParamList | "ε"

ParamList: MoreParams

MoreParams: "," ParamList | "ε"

Const: NUMBERC | STRING | true | false

4. TOKENIZER/LEXER

The Lexer determines which tokens are ultimately sent to the parser and, throw out things that are not defined in the grammar, like comments. Similarly for TinyJs language the Lexer cares about characters (A-Z and the usual symbols), numbers (0-9), characters that define operations (such as +, -, *, and /), open and closed curly brace ({,}), comma, open and closed paternities, semicolons and keywords. The Lexer reads the source, determines which tokens are ultimately sent to the parser. In TinyJs Language, the Tokenizer.java class is responsible for lexing.

5. PARSER

Before Interpretation all input have to be parsed, the parser accept a token from a Lexer (e.g. function, if, return, eval, identifier, number etc.) and parse them to form AST and handles errors if there is a failure. The Parser.java class is recursive descent parser, which means top down parser built from a set of mutually recursive procedures, where for each production rule of the grammar there is one procedure implemented.

6. SYMBOL TABLE AND TYPE CHECKING

The Evaluate.java and Env.java classes handle the type and environment checking. It can register the name of any variable and declared functions with all their value information. Evaluate.java class is where the type checking has been implemented.

7. ABSTRUCT SYNTAX TREE (AST)

The abstract class Node.java represents the building block of the data structure that is used as the Intermediate Representation (IR) for the TinyJs language. The Abstract Syntax Tree (AST) is assembled using the methods provided by the Parser.java class, which is going to be the input for the interpreter. It is started from the main function class TinyJS.java by calling the method of the parser's createAST(). It is up to the parser to create the AST related to a certain TinyJs source code. Each and every Node is distinguished by a label which describes its function and has possibly got a list of children nodes (ArrayList<Node> children).

8. INTERPRETER

The InterpreterNode defined as abstract method in the Node.java class which is capable of running the TinyJs intermediate representation (AST) and produces the correct results.

The interpreter is started from the main function class TinyJS.java by calling the InterpreterNode(new Stack<Env>()) from the Node.java class.

9. TOOLS OF IMPLEMENTATION

- ✓ Netbeans IDE 8.0.2
- ✓ Notepad++

10. REFERENCES

1. Lam, Monica, Ravi Sethi, J. D. Ullman and Alfred Aho. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 2006.
2. Terence Parr: Language Implementation Patterns.
3. Advanced Programming study materials: http://www.di.unipi.it/~giangi/CORSI/AP/AP.html#materiale