SER 502 – Team 6 | Project 2

Aditya Narasimhamurthy
Manit Singh Kalsi
Mohit Kumar
Richa Mittal

Project Pitch

Overview

- Language Design
- Example Programs
- Intermediate code
- Interpretation of IC
- Features of language

Language Design

- Name – RAMM (for obvious reasons!)

- Inspiration
  - Universal Algorithmic Programming Language
  - ALGOL 60
  - Educational enhancements (easier to teach coding)
  - Pseudo code execution (with as minimal changes as possible)

Vision – What our code to looks like

- We want our code to be minimalistic, but also completely readable at the same time.

- Suppose we want to set a variable, say x, to 5. Our code would look something like this:

  ```
  x = 5
  ```

Language Design – RAMM vs. ALGOL 60

- ALGOL Syntax
  - k := 1; // setting a variable k which holds value 1.

- RAMM Syntax
  - K = 1; // setting a variable k which holds value 1.

Intermediate Code Generation

- Once we run the file with our code in it, we need to generate an Intermediate code which can then be fed to our Virtual Machine.

- Assembly like syntax, which makes it easy to read.

- We expect our Intermediate Code (IC) to look like this:

```
SET x 5                    // x = 5
```

Features

- Dynamically Typed
  - Type checks are left until run-time.
- Static scoping
  - A variable always refers to its top-level environment.
- Lazy evaluation
  - Delays evaluation of an expression until its value is needed. Also avoids repeated evaluations.

Project Design

Overview

- Compiler
  - ANTLR
  - Grammar

- Runtime

ANTLR

- Used the latest version – ANTLR v.4.5
- A powerful parser generator for processing/translating structured text or binary files. Used widely to build languages.
- From a grammar, ANTLR generates a parser that can build and walk parse trees.

The Grammar

- Initial objective - easily convert any algorithm/pseudo code to actual code.
- We broke down our programs layer by layer and decided on a flow of execution.
- To give a brief understanding, this is what our grammar looks like at a high level:

The Grammar

- block: (statement | functionDecl)* (Return
  expression)? ;

- statement: assignment | functionCall | ifStatement
  | forStatement |whileStatement ;

- assignment: Identifier indexes? '=' expression ;

- Expression: … |... | Number | …

- Number: Int ('.' Digit*)? ;

- Digit : [0-9];

The Grammar

Say for example, we had our code from the pitch i.e, x = 5.

This is how our grammar would recognize each token:

- Block -> Statement
- Statement -> assignment
- Assignment - > expression
- Expression -> Number
- Number -> Digit
- Digit -> [0,1,2,3,4,5,6,7,8,9]

Grammar Features

- Our grammar supports data types like boolean and number.
- Data structures supported include lists.
- Control constructs provided are – if and while
- Can handle math operators like +, -, *, /, %
- Handles relational operators like <, <=, >, >=, ==, !=

Intermediate Code

- Our Intermediate code implements prefix notation.

- For example, if we take our code a = 5, the intermediate code would look like:

  SET A 5

- We have also used our own notations for comparisons and looping constructs.

  Here's a concise representation of some of most used notations:

Intermediate Code

Action	Notation
>	GT
<	LT
>=	GE
<=	LE
==	E
!=	NE
if()	CHECK
While()	LOOP
Function call	LOAD

Runtime - Features

- RAMM's Runtime is completely based on JAVA.
- It is dynamically typed.
- It internally uses stacks, linked lists and hashmaps
- The intermediate code is in prefix notation and the runtime is designed accordingly to process prefix expressions

Runtime - Features

- A symbol table is a data structure used by a compiler where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.
- During execution, every function call is recorded and maintained on a stack.
- In RAMM, the symbol table is implemented as a linked list of hash maps.

Project Implementation

Lexer and Parser

- Lexical analysis is handled by the Lexer, where it separates a stream of characters into different words or 'tokens‘.

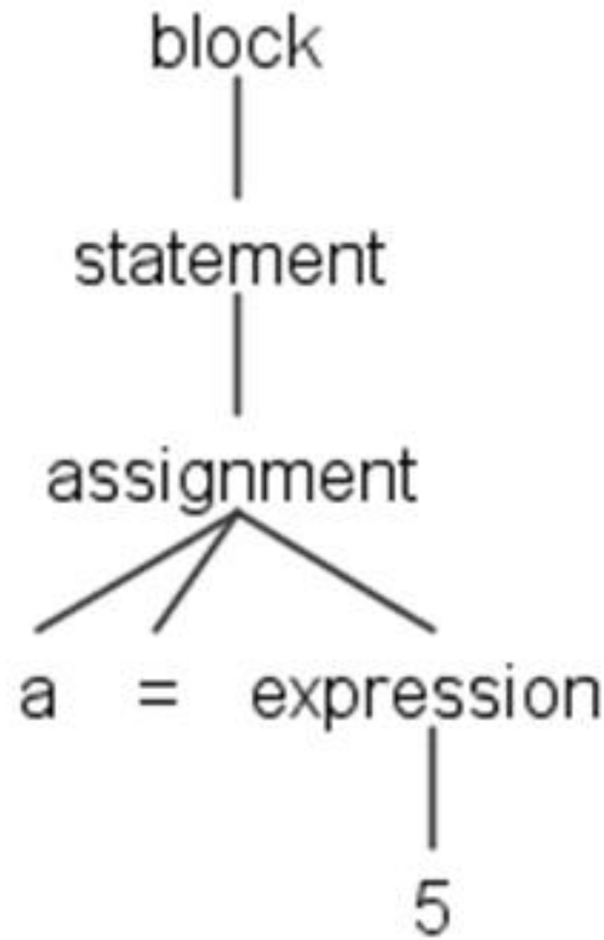
- Syntactical analysis is handled by the Parser. It receives the tokens/ input from the Lexer in the form of sequential source program instructions and breaks them up into parts defined in our grammar.

Abstract syntax tree

- This is a tree representation of the abstract syntactic structure of source code written in a programming language.
- Each node of the tree denotes a construct occurring in the source code.
- The syntax is "abstract" in not representing every detail appearing in the real syntax.

- For our example, the AST would look something like this:

Abstract syntax tree



Abstract syntax tree

- Using this tree, our next goal is to reach/ walk to each individual node programmatically and generate the intermediate byte code.

- Made use of an inbuilt data type in ANTLR called ParseTree to parse each individual leaf/ non-leaf element.

- Once every element was parsed, we created the intermediate byte code, which was then fed to the VM for execution.

Abstract syntax tree

- By default, ANTLR generates a parse-tree listener interface that responds to events triggered by the built-in tree walker. .
- To walk a tree and trigger calls into a listener, ANTLR’s runtime provides the class ParseTreeWalker.
- To make a language application, we write a ParseTreeListener.

Abstract syntax tree

- There are situations, however, where we want to control the walk itself, explicitly calling methods to visit children.
- Option -visitor asks ANTLR to generate a visitor interface from a grammar with a visit method per rule.
- The key “interface” between the grammar and our listener object is called JavaListener, and ANTLR automatically generates it for us.
- It defines all of the methods that the class ParseTreeWalker from ANTLR’s runtime can trigger as it traverses the parse tree

Runtime - Implementation

- Internally uses stacks, linkedlists and hashmaps to maintain the environment, symbol table and activation records
- A symbol table is a data structure used by a compiler where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.
- During execution, every function call is recorded and maintained on a stack.
- In RAMM, the symbol table is implemented as a linked list of hash maps.