Overview

The goal of this project is to modify the Core interpreter from Project 3 to now handle

function definitions and function calls.

Your submission should compile and run in the standard environment on stdlinux. If you

work in some other environment, it is your responsibility to port your code to stdlinux and

make sure it works there.

You will need to modify the parser and executor functions to accommodate function calls.

You should not need to modify the scanner for this project, but you are allowed to so

if you want. You also do not need to modify the existing printer functions or add printer

functions for the new parts of the grammar, but it might help you check your work and

troubleshoot bugs if you do implement print functions for the new classes.

Functions

On the last page is the modified Core grammar to allow functions. Here is an example of a

Core program that defines and calls a function:

A white page with black text

Description automatically generated

For simplicity you may assume functions will only accept array variables as actual pa-

rameters, and our parameter passing will be done with call by sharing - x and a point to

the same record, y and b point to the same record. In this example, as output we should

see 3, then 1, then 2.

Your interpreter should support recursion. This will require implementing a call stack.

1

Input to your interpreter

The input to the interpreter will be same as the input from Project 3, a .code and a .data

\_le

Output from your interpreter

All output should go to stdout. This includes error messages - do not print to stderr.

The parser functions should only produce output in the case of a syntax error. You

should implement checks to ensure the modified grammar is being followed.

For the executor, each Core output statement should produce an integer printed on a

new line, without any spaces/tabs before or after it. Other than that, the executor functions

should only have output if there is an error.

Invalid Input - Syntax Checks

For this project you will need to modify some existing parse functions, and add new classes

and new parse functions. Your parse functions should ensure the syntax rules of the modified

grammar are being followed.

Invalid Input - Semantic Checks

With the addition of functions to the language you will need to perform some semantic

checks. You DO NOT need to include the semantic checks from project 2, you only need to

handle the new ones described here. You can perform the semantic checks during parsing,

during execution, or separately.

Here are the semantic rules you should be checking:

1. Every function should have a unique name (no overloading).

2. Each function call has a valid target.

3. Your parser should check that the formal parameters of a function are distinct from

each other. You do not need to implement checks to verify the arguments are all

distinct or are all array variables, we will consider using integer variables as arguments

as undefined by the language.

Implementation Suggestions

Here are my suggestions for how to implement the project:

1. Focus first on modifying/creating parse functions, and add new print functions to verify

your modified parser. There is no point in even thinking about the execution unless

you are confident in your parse tree!

2

2. You will need a stack of \frames" to allow recursion. A frame should be the same as

your local memory from project 3. You then need a stack of frames, so if you used my

suggestions for project 3 then for project 4 you will have a stack of stack of maps!

3. Implement the semantic checks last.

4. During execution you will need a way to \jump" across the tree, or in other words

when I am in the StmtSeq and I want to call a function named \Foo", I need some

way of finding the part of the tree where \Foo" is defined in the DeclSeq.

5. The hardest part will be implementing the execute function for the function call. To

execute a function call, you need to:

(a) Create a new frame

(b) Create the formal parameters in this frame and copy over the values of the argu-

ments

(c) Push the frame to the top of the stack

(d) Execute the body (StmtSeq) of the function

(e) Pop the frame o\_ the stack

Testing Your Project

I will provide some test cases. The test cases I will provide are rather weak. You should do

additional testing testing with your own cases. Like the previous projects, I will provide a

tester.sh script to help automate your testing.  
  
Please note this is a language like C or Java where whitespaces have no meaning, and whitespace can be

inserted between keywords, identifiers, constants, and specials to accommodate programmer style. This

grammar does not include formal rules about whitespace because that would add immense clutter.

Recall epsilon is the empty string.

<procedure> ::= procedure ID is <decl-seq> begin <stmt-seq> end

| procedure ID is begin <stmt-seq> end

<decl-seq> ::= <decl > | <decl><decl-seq> | <function> | <function><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= <decl-integer> | <decl-array>

<decl-integer> ::= integer id ;

<decl-array> ::= array id ;

<function> ::= procedure ID ( <parameters> ) is <stmt-seq> end

<parameters> ::= ID | ID , <parameters>

<stmt> ::= <assign> | <if> | <loop> | <out> | <in> | <decl> | <call>

<call> ::= begin ID ( <parameters> ) ;

<assign> ::= id := <expr> ; | id [ <expr> ] := <expr> ; | id := new integer [ <expr> ]; | id := array id ;

<out> ::= out ( <expr> ) ;

<in> ::= in ( id ) ;

<if> ::= if <cond> then <stmt-seq> end

| if <cond> then <stmt-seq> else <stmt-seq> end

<loop> ::= while <cond> do <stmt-seq> end

<cond> ::= <cmpr> | not <cond> | <cmpr> or <cond> | <cmpr> and <cond>

<cmpr> ::= <expr> = <expr> | <expr> < <expr>

<expr> ::= <term> | <term> + <expr> | <term> – <expr>

<term> ::= <factor> | <factor> \* <term> | <factor> / <term>

<factor> ::= id | id [ <expr> ] | const | ( <expr> )