Project Milestone 2 part A

Abhi Agarwal

My project two documentation is organized in a way where I construct the table in the beginning, and then break down the details I'm going to present and explain them in detail (for documentation and understanding). I have also attached definitions of word I'm using at the bottom to reduce ambiguity and misunderstandings. I tried to simplify my table by using function calls to test the different types, and that's something I think is a limitation of my Syntax Directed Definition, but I think it was important for me to do that. I have an understanding of how to implement these functions in Hacs.

Syntax Directed Definition Table

Production	Semantic Rules
$AProg \rightarrow ABlock_1$	ABlock.inScope = top; $ABlock_1.s = \{\}$;
$ABlock \rightarrow C_1;\ ABlock_1$ $\mid F_1;\ ABlock_1$ $\mid \epsilon$	$C_1.s = ABlock.s; ABlock_1.s = C_1.s$ $F_1.s = ABlock.s; ABlock_1.s = F_1.s$
$C \rightarrow class\, id\, \{CS_1\}$	$C.s = Extend(C.s, id.sym, class)$; $C.inScope = class; CS_1.s = C.s$;
$F \rightarrow function\, T_1\, id(P_2)\, \{S_3\}$	$F.s = Extend(F.s, id.sym, function)$; $F.inScope = function; S_3.s = F.s$;
$M \rightarrow T_1\, id(P_2)\, \{S_3\}$	$M.s = Extend(M.s, id.sym, method)$; $M.inScope = method; S_3.s = M.s$;
$CS \rightarrow S_1; CS_2$ $\mid M_1; CS_2$ $\mid \epsilon$	$S_1.s = CS.s; CS_2.s = S_1.s$ $M_1.s = CS.s; CS_2.s = M_1.s$
$S \rightarrow T_1\, id;$ $\mid if(E_1)\, \{S_2\}$ $\mid while(E_1)\, \{S_2\}$	$T_1.s = S.s;\ S.s = Extend(S.s, id.sym, T_1.type)$ $E_1.s = S.s$; checkIfBoolean $(E_1.type)$; $S_2.s = S.s$ $E_1.s = S.s$; checkIfBoolean $(E_1.type)$; $S_2.s = S.s$
$P \rightarrow T_1\, id, ; P_2;$ $\mid T_1\, id;$	$T_1.s = P.s;\ P.s = Extend(P.s, id.sym, T_1.type)$; $P_2.s = P.s$; $T_1.s = P.s;\ P.s = Extend(P.s, id.sym, T_1.type)$
$E \rightarrow this.E_1;$ $\mid E_1.m$ $\mid !E_1$ $\mid -E_1$ $\mid +E_1$ $\mid -E_1$ $\mid E_1 * E_2$ $\mid E_1/E_2$ $\mid E_1 - E_2$ $\mid E_1\%E_2$ $\mid E_1 + E_2$ $\mid E_1 < E_2$ $\mid E_1 <= E_2$ $\mid E_1 > E_2$ $\mid E_1 >= E_2$	$classScope(E.inScope); E_1.s = E.s$ $E_1.s = E.s; checkIfObject(E_1.type)$; $E_1.s = E.s; checkIfBoolean(E_1.type)$; $E.type = E_1.type$; $E_1.s = E.s; checkIfInt(E_1.type); E.type = E_1.type$; $E_1.s = E.s; checkIfInt(E_1.type); E.type = E_1.type$; $E_1.s = E.s; checkIfInt(E_1.type); E.type = E_1.type$; $E_1.s = E.s; E_2.s = E.s$; $checkIfInt(E_1.type, E_2.type); E.type = E_1.type$; $E_1.s = E.s; E_2.s = E.s$; $checkIfInt(E_1.type, E_2.type); E.type = E_1.type$; $E_1.s = E.s; E_2.s = E.s$; $checkIfInt(E_1.type, E_2.type); E.type = E_1.type$; $E_1.s = E.s; E_2.s = E.s$; $checkIfInt(E_1.type, E_2.type); E.type = E_1.type$; $E_1.s = E.s; E_2.s = E.s$; $checkSameType(E_1.type, E_2.type); E.type = E_1.type$; $E_1.s = E.s; E_2.s = E.s$; $checkSameType(E_1.type, E_2.type)$; $checkIntOrString(E_1.type, E_2.type); E.type = boolean$; $E_1.s = E.s; E_2.s = E.s$; $checkSameType(E_1.type, E_2.type)$; $checkIntOrString(E_1.type, E_2.type); E.type = boolean$; $E_1.s = E.s; E_2.s = E.s$; $checkSameType(E_1.type, E_2.type)$; $checkIntOrString(E_1.type, E_2.type); E.type = boolean$; $E_1.s = E.s; E_2.s = E.s$; $checkSameType(E_1.type, E_2.type)$; $checkIntOrString(E_1.type, E_2.type); E.type = boolean$;

Production	Semantic Rules
$\mid E_1 == E_2$	$E_1.s = E.s; E_2.s = E.s;$ $checkSameType(E_1.type, E_2.type); E.type = boolean;$
$\mid E_1 != E_2$	$E_1.s = E.s; E_2.s = E.s;$ $checkSameType(E_1.type, E_2.type); E.type = boolean;$
$\mid E_1 \&\& E_2$	$E_1.s = E.s; E_2.s = E.s;$ $checkifBoolean(E_1.type, E_2.type); E.type = boolean;$
$\mid E_1 \| E_2$	$E_1.s = E.s; E_2.s = E.s;$ $checkifBoolean(E_1.type, E_2.type); E.type = boolean;$
$\mid E_1 += E_2$	$E_1 = E.s; E_2.s = E.s;$
$\mid E_1.length$	$E_1.s = E.s; checkIfString(E_1.type);$ $E.type = E_1.type$
$\mid E_1.charCodeAt(T_2id)$	$E_1.s = E.s; checkIfString(E_1.type); T_2.s = E_1.s;$ $checkIfInt(T_2.type); E.type = E_1.type$
$\mid E_1.charCodeAt(T_2id, T_3id)$	$E_1.s = E.s; checkIfString(E_1.type);$ $T_2.s = E_1.s; T_3 = E_1.s;$ $checkIfInt(T_2.type, T_3.type); E.type = E_1.type$
$\mid L_1$	$L_1.s = E.s; L.type = L_1.type;$
$OL \rightarrow id : L$	
$L \rightarrow id$	$L.type = Lookup(L.s, id.sym)$
$\mid Integer$	$L.type = int$
$\mid String$	$L.type = string$
$\mid Object$	$L.type = object$
$T \rightarrow integer$	$T.type = int$
$\mid string$	$T.type = string$

1 - Limitations and Decisions

1. Inherited Attributes: s, Synthesized Attributes: inScope, Functions: checkifBoolean, Extend, Lookup, classScope, checkIfInt, checkIfIntOrBoolean, checkSameType

2. Declarations: **AProg** is the beginning of the program, **ABlock** is a block (either a function or a class), **C** is a class, **F** is a function, **M** is a member function, **CS** is a class statement (either a statement or a member function), **S** is a statement, **E** is an expression, **L** is a literal, **T** is a type, **OL** is an object literal definition, and **P** is a parameter check.

3. My way around knowing the scope

4. The block is each module of the particular program where module could be function, or a class.

5. For this particular implementation there is no support for global variables, but I have thought about it. It would be a part of the modules that I have described above, and I would extend the modules to be: function, class, or global variables.

6. For an object declaration `var` is necessary as suggested in the example.

7. Two passes through the code: first to declare and get all the information, and second to validate it.

8. Declaration of many functions to check if they are the correct type. They are straight forward, and do what the function names are. They are very similar to the safeCast() function we created for HW6. For example: checkIfBoolean(...) checks if a particular type is boolean - if it is then return Error();

9. My implementation I extend a class declaration into the Class scope. This is because I'm now able to check if I'm inside a class or not by looking through the immediate scoped variables.

10. My implementation of the string and its additional operators are included within the Expressions (E).

2 - Extra notes

11. To me (my assumption is that) `any` means that we're able to convert between types very explicitly without using a cast, and we're able to assign certain values regardless of the type. Moreover, the every type is assignment compatible with itself is fairly straight forward, and the standard case.

12. Identifier literals: must exist within the scope where it is declared, and has to have the correct type. We have to do a type check to see if the assignment is the same as the type it was declared as.

13. Int literals: We've to check if they've the type int to classify them as Integers. This is just mechanical as we've to simply check if their type matches the type `int`.

14. Object literals: String literals: We've to check if they've the type string to classify them as `string`.

15. Object literals: We must check if m_k for each k to see if it is not already within the current scope, and if it is then we have to generate an error as they must be distinct.

16. Object literals: We also have to check from the scope of the class to make sure the m_k exists in the class type.

17. l-value: We also have to check for the type of l_k, for a given k, and if it has a type which is assignment compatible to the type declared for the member then it's been validated.

18. l-value: We have to check whether the lexeme on the left site is either an Identifier or an instance of a class variable for which the element it is trying to access exists, and if either of these conditions are not present then we return an error.

19. l-value: We have to check whether the lexeme on the left site is either an Identifier or an instance of a class variable for which the element it is trying to access exists, and if either of these conditions are not present then we return an error.

20. Expressions: We have to check if the scope we're in is in fact a class block, and if it is then we have to check if the variable has been defined in the local nested scope. If the variable hasn't been defined then we throw an error because it can't be referenced.

21. this: We have to check if the scope we're in is in fact a class block, and if it is then we have to check if the variable has been defined in the local nested scope. If the variable hasn't been defined then we throw an error because it can't be referenced.

Definitions

Skip over this section, but here I've defined some things I think are important, and I wanted to have the definitions here to reduce ambiguity and misunderstandings.
Scope: Tied to program "blocks" (lexical) vs runtime stack.
Global scope: Scope that surrounds a block where the block can be a function, variable, object, or class.
Top scope: The scope of the whole program that includes all declarations of classes, functions, and global variables that are declared in the main script.
Current scope: For a given block we're currently in, all the local functions/variables/classes we've encountered.
New scope: We're going into a new program block, and therefore we declare a new nested scope including values from the global scope.