Milestone 2

Zoe Guan, Samuel Laferriere, Emil Rose

March 6, 2015

Design Decisions

For the symbol table, we started off using a symbol stack of scopes which held both variable declarations and type declarations. However, this proved hard to work with when trying to differentiate between a type alias and a variable of that type alias, so we ended up using a separate symbol table for type declarations, which although perhaps not the prettiest of solutions, made our life much easier. The main difficulty in implementing the symbol table using SableCC was the fact that the CaseANodeName methods of the DepthFirstAdapter all return void. When trying to declare a variable of a struct type for example, we needed access to the types declared inside the struct, and so a non void return would have been helpful. To circumvent this problem, we used a global variable which held the most recent visited node's Type.

We represented and compared types using type classes (BooleanType, AliasType, etc storing the relevant information for each type). For the typechecker, we took advantage of the outANodeName methods in the SableCC DepthFirstAdapter. These methods are always called after the node's children are explored, so we just overrode these methods, guaranteeing that the node would have the type information required about its children. Since our compiler does symbol table construction and type checking in one pass, for some nodes that involve both updating the symbol table and typechecking the children (ex: variable declaration with non-empty right side, function declaration), we made use of both the inANodeName and outANodeName methods (ex: for function declarations, we add the mapping to the symbol table before typechecking the children because our typecheck of return statements requires the function type). We also used the inANodeName methods for opening scopes. Type information was stored in a hashmap from nodes to types, allowing efficient access, even for large programs with many nodes. To keep the code at a reasonable length, we heavily used helper methods for the type checker because of how repetitive many of the cases are.

Scoping Rules

- all top-level declarations belong to the outermost scope
- in a function declaration, the list of parameters (possibly empty) opens a new scope that closes at the end of the function body
- the start of a non-function-body block opens a new scope that closes at the end of the block

  - in our implementation, each of the following things corresponds to a block node in the AST
    * a function body (but a function body does not open a new scope because it is in the same scope as the list of parameters)
    * a for loop body
    * an if statement body
    * a case or default body in a switch statement (a case or default body is an implicit block without braces)
    * any other block of statements

- a for loop `init` statement opens a new scope that closes at the end of the for loop body
- an if statement `init` statement opens a new scope that closes at the end of the if statement body
- a switch statement `init` statement opens a new scope that closes at the end of the switch statement body
- a struct type declaration body opens a new scope that closes at the end of the body

Type Errors

Declaration, Statement or Expression Type	**Error**	**Program**
Variable declaration (1.1)	- variable name already declared in current scope	1.1_var_decl.go
Type declaration (1.2)	- type name already declared in current scope	1.2_type_decl.go
Function declaration (1.3)	- function name already declared in current scope - function has two parameters with the same name - function returns a value but there is an execution path without a return statement	1.3_func_decl.go 1.3_func_decl_2.go 1.3_func_decl_3.go
Return statement (2.4)	- enclosing function has a return type but return statement has no expression - return expression is not of same type as return type of enclosing function	2.4_return.go 2.4_return_2.go
Short declaration (2.5)	- there are no new variables on left-hand side - there is a previously declared variable on the left-hand side whose type does not match the type of its assigned expression - two variables on the left-hand side have the same name	2.5_short_decl.go 2.5_short_decl_2.go 2.5_short_decl_3.go
Assignment (2.7)	- lvalue and its assigned expression have mismatched types	2.7_assign.go
Op-assignment (2.8)	- arguments to += are neither numeric nor string or have mismatched types - arguments to −=, ∗=, /=, %= are non-numeric or have mismatched types - arguments to one of the remaining assignment operators are not integers or have mismatched types	2.8_op_assign.go 2.8_op_assign_2.go 2.8_op_assign_3.go
Print statement (2.10)	- `print` or `println` expression is not of type int, float64, bool, rune, string or an alias of one of those	2.10_print.go
For loop (2.11)	- while expression does not have type bool - for expression in three-part for loop does not have type bool	2.11_while.go 2.11_for.go
If statement (2.12)	- if expression must have type bool	2.12_if.go
Switch statement (2.13)	- type of switch expression does not match type of case expressions - switch statement has no expression but case expressions do not have type bool	2.13_switch.go 2.13_switch_2.go
Identifier (3.2)	- usage of undeclared identifier	3.2_id.go

continued on next page

Declaration, Statement or Expression Type	**Error**	**Program**
Unary expression (3.3)	- in +`expr` or −`expr`, `expr` does not have type int/float64/rune - in !`expr`, `expr` does not have type bool - in ^`expr`, `expr` does not have type int/rune	3.3_unop.go 3.3_unop_2.go 3.3_unop_3.go
Binary expression (3.4)	- one of the arguments to ‖ or && is not a bool - arguments to == or != are not comparable - arguments to <, <=, >, or >= are not ordered - arguments to + are neither numeric nor string or have mismatched types - arguments to −, ∗, /, % are non-numeric or have mismatched types - arguments to |, &, &^, ^ are not integers or have mismatched types	3.4_binop.go 3.4_binop_2.go 3.4_binop_3.go 3.4_binop_4.go 3.4_binop_5.go 3.4_binop_6.go
Function call (3.5)	- in `expr(arg1,...,argn)`, `expr` does not have type `(T1*...*Tn)` → `T` - in `expr(arg1,...,argn)`, the types of the arguments do not match the types in the function declaration	3.5_func_call.go 3.5_func_call_2.go
Indexing (3.6)	- in `expr[index]`, `expr` is neither a slice nor an array - in `expr[index]`, `index` is not an int	3.6_indexing.go 3.6_indexing_2.go
Field selection (3.7)	- in `expr.id`, `expr` is not a struct - in `expr.id`, `expr` does not have a field called `id`	3.6_indexing.go 3.6_indexing_2.go
Append (3.8)	- in `append(id, expr)`, `id` is not a slice - in `append(id, expr)`, `expr` does not have the same type as elements of `id`	3.7_selection.go 3.7_selection_2.go
Type cast (3.9)	- in `type(expr)`, `type` is not of type int, float64, bool, rune, or an alias of one of those four - in `type(expr)`, the type of `expr` is not of type int, float64, bool, rune, or an alias of one of those four	3.9_type_cast.go 3.9_type_cast_2.go

Comparability for Binary Expressions

- bool and bool
- rune and rune
- int and int
- float64 and float64
- string and string

- struct and struct: the struct types must be identical (must correspond to the same type declaration) and all their fields must be comparable
- array and array: the arrays must have the same size and the element types must be comparable
- aliased type and aliased type: the aliased types be identical (must correspond to the same type declaration) and the underlying types must be comparable

Contributions

Samuel wrote the symbol table, and Emil wrote the type checker. Zoe wrote the test programs and most of this document, debugged the symbol table and type checker, and built a large test suite to make sure no errors fell through.