Project Milestone 2

Kevin Ayuque (kja306), Goktug Saatcioglu (gs2417)

April 2019

1 Introduction

This is the documentation for Milestone 2 of the Spring 2019 Compiler Construction course. We had worked together on Milestone 1 and continue working together for Milestone 2. The choices made and issues resolved is outlined in Section 2. Our explanation of functions we defined for the SDD is given in Section 3. Section 4 has our work log which shows who worked on what part of the project. Finally, Section 5 is the actual SDD table for all the rules in the grammar given in the assignment specification. We also assume very basic knowledge of functional programming syntax (i.e. OCaml or SML) throughout the document along with implementation of some assumed functions in OCaml.

2 Choices Made and Issues Resolved

2.1 Attributes Used

Throughout the SDD we use the following attributes:

- *env* - This attribute refers to the referencing current environment of a non-terminal symbol in the grammar, i.e. *env* is the symbol table up to and including this non-terminal symbol.
- *fun* - This attribute is only for function declarations and stores the tuple (type, name, argument list). This information is used such that the top level environment for the rule starting with symbol `P` can be updated properly.
- *rets* - This attribute holds what type a given statement list should return. It is determined when a function is declared, i.e. the declaration production of the grammar, and then passed onto each statement after the declaration such that if we ever encounter return statement we can check whether there is a type mismatch or not.
- *retr* - This attribute determines whether a given rule derivation eventually has a `return` statement inside it. The reason this is necessary is because we need to make sure that every function actually returns something. So, once we check all the statements we can report an error at the function declaration level if a return statement is missing.
- *type* - This attribute holds the typing information for an expression. It is updated and checked for as the assignment sheet specifies such that we get correct type checking behavior.
- *isLVal* - This synthesized attribute is only for assignments to make sure we only make assignment to l-values which are either **id**'s or ***id**'s (i.e. pointer dereferences of identifiers). If an expression production derives an l-value immediately the attribute is is true and otherwise it is false. Naturally, it is only attached to expressions.

2.2 Dependency/Information Flow

The type checker works structurally on the program, i.e. it uses the grammar to evaluate smaller sub-expressions and obtain type information for bigger expressions. For expressions we have the inherited attribute *type* that refers to the type of the expression and uses the typing information of the sub-expressions to determine the type of larger expressions. While determining the types the semantic rules also check for any errors and report them if seen by using type-checking functions defined in the Functions Defined section. To properly type check we also use the synthesized attribute *env* that passes the relevant environment down to sub-expressions. This is necessary so that all expressions have access to a symbol table with the correct referencing environment for that expression. We mainly use *env* in expression grammar productions to check if names being used have been declared and if functions are being properly called, i.e. whether there are type-mistmatches between the actuals and the formals of a function for a function call. For statements the attribute *env* works similarly but this time it is inherited. This is necessary because symbol table information must be passed onto expressions inside statements and the symbol table must be updated if a new variable is declared. We also check whether a variable has already been declared and give an error if so. There is also another inherited attribute called *rets* that passes information of what type the function that encloses these statement has to return. If we ever encounter a `return` statement then we can check whether we are returning the proper type. We also use the synthesized attribute *retr* to make sure a list of statements eventually returns something. This is necessary as with the current grammar we can actually write programs with functions that declare a return type but they never actually return anything. So as we build the grammar, we can make sure that a return statement is included somewhere. For function declarations we again have *rets* and *env* which passes on to the statements underneath the function declaration what must be returned and what variables have been declared in the function argument. It also has *retr* and we check at this level whether we have actually returned something. Furthermore, there is the synthesized attribute *fun* that basically allows the program environment to obtain information about all of the declarations. Also, the program grammar rule also has an environment `env` which we use to update the global symbol table and pass it onto the function declarations. This global environment assumes that `GlobalPreDefs` is already given as specified in the assignment sheet and updates it using Di.*fun* for all Di such that we get a symbol table of function declarations. Then we check this table for if a main function has been declared and if it has been then whether it has been properly declared as per the assignment specification. The descriptions of functions related to the environment/symbol-table checking and program checking can be found in the Functions Defined section. Finally, we have one last synthesized attribute called *isLVal* which is true if an expression production immediately derives a l-value and otherwise false. This is so that we make assignments only to identifiers and the existence of the identifier is checked on the expression level. Overall, our implementation is pretty straightforward but also has some assumptions which we discuss next.

2.3 Assumptions and Choices

We had to make a few simplifying assumptions and these are as follows:

- Scoping: We assume that the scope of a top-level variable extends to all statements and nested statements below it. That is if we have statements of the form { `int x; ...` { `...` { `...` } `...` } `...` } then the variable `x` is valid for all nested statement lists and cannot be re-declared. We did this because it is a simple implementation and the assignment specification gives no rules on scoping meaning we were free to choose what to do. Better scoping rules can possibly be implemented with the SDD and the Dragon Book has a good discussion on this. Overall, our implementation gives us static scoping where the lifetime of a variable is from the point it is declared inside a function declaration until the function ends. Note that function arguments also follow this rule and thus we cannot declare a variable with the same name as a function argument. Furthermore, statement lists that introduce a new block do have a local scope so they cannot hide top-level declarations. Finally, the symbol table is easy to implement and we assume that we create copies of existing tables whenever we make an assignment in the semantic rules. This way if there is hiding of the scope of a variable we can recover the hidden variable once the hiding ends. A simple hash map should work well in this situation. However, the copying of tables is very inefficient and something we can improve upon.

- Function names: The only time our scoping rules change is for function declarations. We see that all functions are aware of other functions meaning the order of function declarations does not matter. However, we require unique function names and these are checked in the order they are declared. This could be considered a quirk of our language but it is actually consistent with the way we define scoping and the way we would like functions to be called, i.e. inside any function we should be able to call any other function. Note that our rules for function names means that direct method overloading or overriding are not allowed but such object-oriented programming features can be implemented by emulating dynamic dispatch (which is also how OOP in standard C would be implemented).

- Function return types: The functions must return either a type int or *char and can never return a type void. This is because we have no global variables in our programs meaning it is pointless to call a function for its side effects as it is not possible to affect a global state due to lack of global variables. Thus, we impose this restriction to make our type checking of whether a function actually returns something easier. The checks for whether a function body actually returns something is discussed next.

- Checks for whether a function body actually returns something: This is a peculiar issue since the grammar allows us to actually build functions that have no return statements however, as discussed above, we actually require that functions return something. To resolve this issue we use the synthesized attribute *retr* which naturally ends up restricting the number of valid programs to those that are guaranteed to return something. If we see a if-body rule followed by more statements we require that the more statements return something. If we see a if-body-else-body rule followed by more statements we either require both if-body and else-body to return something or the more statements to return something. If we see a while-body followed by more statements we require that the more statements return something. Finally, for statement lists we require that either the nested list or the rest return something and this may seem counter-intuitive but the idea is simple. At the top-level without any conditionals we can have an arbitrary amount of nesting before we get a return and this will still be good. If we have conditionals then our conditional rules will make sure the program is good. For every other production we simply propagate the information back up the tree such that at the function declaration level we can then do one last check and return an error if we have a bad program. This checking has the additional benefit that we are guaranteed that statements such as `int x; x = foo(5,6);` actually assign an `int` to variable `x` because they have passed the type checker. In a way, this implementation of MiniC resembles functional languages as there is no state on the global level but we also have imperative features as we have state on the local level.

- Global variables: As the grammar has no way to introduce global variable declarations we safely assume that there are no global variables declared throughout our SDD implementation. This is not a major issue but nonetheless merits mentioning.

- `GlobalPreDefs`: We assume that `GlobalPreDefs` is the global predefined functions as given in the assignment sheet and are in such a format that we can create an initial environment for the program. That is, `GlobalPreDefs` is in a format such that a symbol table can be easily created from it. This is not too big of an assumption but must be noted. Alternatively, we could have just assumed some sort of pre-processor appends all pre-defined functions to any created program before beginning the lexing phase so our analysis would work without initializing with some `GlobalPreDefs`. That is we could simply run the type checker and collect this information as analyze the program. We take the former approach of assuming the `GlobalPreDefs` variable but the other approach works too.

- Expression assignments at the statement level: We also noticed that the grammar itself does not prevent assignment such as `1 + 2 + 3 = 4 + 5 + 6;` or `1 < 2 = 2 < 1;`. Of course such assignments are nonsensical so we must make sure that in an assignment statement the left-side should be an l-value. So we must check whether the left side of an assignment expression is either an identifier or a pointer dereference symbol immediately followed by an identifier. This is where the *isLVal* synthesized attribute comes into play and is declared false for all other expression productions other than the two cases described above. So if the expression gives us an identifier this attribute becomes true and if it the expressions gives us a pointer dereference then it is true as long as the expression following the dereference symbol gives us an identifier. This attribute allows us to resolve the non-valid assignment issues and we can check at the statement level whether the assignment is valid by accessing the *isLVal* attribute of the left side of the assignment. Furthermore, we check whether an identifier has been declared at the expression level allowing us to ensure that an assignment is made to a l-value and the variable being assigned to has been declared.

- Assumed behavior: We check for an error before we do any updating to the attributes. This is necessary as we wish the error to be checked before running some function that assumes no errors will occur. Otherwise, we will obtain erroneous behavior. While this assumption is implicit we chose to explicitly state it here. Furthermore, we assume that if there is an error then the error function correctly handles this as to either continue analyzing the program or reporting an error with a location arrow. The assignment specification does not specify how to handle errors or what happens to the analysis in the case of an error so we leave this as open-ended issue. A user, in theory, can provide implementations of these error functions so as to achieve their desired behavior.

- Assumed functions: We assume a variety of easily implementable functions along with an implementation of a symbol-table that works nicely with our assumed functions. This assumption is further discussed in the Functions Defined section. As a final note, we push the checking of whether a proper main has been declared into our assumed functions which greatly simplifies this task.

2.4 Issues Resolved

The preceding sections describe all the issues we encountered and how we resolved them such as by making assumptions or making choices.

3 Functions Defined

3.1 Type-checking functions

The functions below can be implemented using simple pattern matching on ADTs or other programming constructs. The implementation is easy and up to the user so we abstract away the concrete implementation but describe what each function should do.

- `isEqual(type, type)`

  Returns `true` if both types are equal. Otherwise it returns `false`.

- `isInt(type)`

  Returns `true` if `type` is an `int`. Otherwise it returns `false`.

- `isPointer(type)`

  Returns `true` if `type` is a pointer. Otherwise it returns `false`.

- `isIntOrPointer(type)`

  Returns `true` if `type` is an `int` or a pointer. Otherwise it returns `false`.

- `toString(type)`

  Transforms the `type` to a `string` — i.e., type `int` becomes "int".

3.2 Environment functions

Here we assume that the environment is a symbol table. Furthermore, we assume that symbol table look ups and additions can be easily done and are already given with the symbol table implementation. Again, we abstract away the implementation but stick to our functional programming mindset such that the symbol table is never mutated but a new updated copy is returned. This is inefficient but has the benefit of being side-effect free so we know that some function call elsewhere in our SDD won't inadvertedly change some other state of the table.

- `existsEnv(env,name)`

  Performs a lookup on the `env` symbol table and returns `true` if a `name` symbol exists. Otherwise it returns `false`.

- `extendEnv(env,type,name)`

  Creates and returns a new environment where `new_env = env + {(type, name)}`. We can pass `env = {}` if we have no prior environment. However, it is not possible to create a blank environment, i.e. the empty symbol table.

- `makeEnv(fun_decls)`

  Creates and returns a new environment using function declarations, we commonly initialize this with the `GlobalPreDefs` which is specified in the specifications sheet. We can make a blank environment here by passing `env = {}` here.

- `getFunType(env,name)`

  Performs a lookup on the `env` symbol table and returns the type of the function that matches `name`.

- `checkArgTypes(env,name,(actuals list))`

  Takes an environment `env` and the `name` of the function being called and returns `true` if the actual types matches the formal types of the function being called. Otherwise it returns `false`.

- `getVarType(env,name)`

  Returns the type of a variable `name` by performing a lookup on the `env` Symbol table. If the name does not exist in the symbol table it will return `None`, i.e. we use the functional programming `Option` type to indicate it either returns something or returns nothing.

3.3 Program functions

Again the implementation are abstracted but these functions are easy to implement. Given an environment (i.e. symbol table) we can iterate over it to check for certain properties such as has a main function been declared. If we find the property we return true and otherwise we return false.

- `existsMain(env)`

  Takes an environment `env` and returns `true` if a `main` function has been declared on the respective environment. Otherwise it returns `false`.

- `checkMainRetType(env)`

  Takes an environment `env` and returns `true` if the `main` function return type is `int`. Otherwise it returns `false`. If `main` does not exist it also returns `false`.

- `checkMainArgs(env)`

  Takes an environment `env` and returns `true` if the `main` function arguments are all of type `*char`. Otherwise it returns `false`. If `main` does not exist it also returns `false`.

4 Work log

Date	Details	Description
March 15	Met up to discuss the assignment specification. Agreed on the attributes *env*, *rets*, *type*. Agreed to think and meet up next week.	Expression Types
March 22	Kevin will work on expressions. Goktug will work on statements. Agreed to use Overleaf and continue using Slack. Will resolve issues as they come up Meet up Friday April 5	Program Declaration Statements Expressions
April 5	Met up to discuss our implementations. Considered function declarations and environments. Discussed issue of returns. Agreed to use the attributes *retr* and *fun*. Goktug will finish the SDD with our agreements. Kevin will start working on the document. Goktug will finish editing the SDD.	Further discussion Issue of return Typing up the document
April 12	Went over the document. Finished the write-up. Proof-read together. We will proof-read one more time before submitting.	Finishing touches
April 15	Realized the issue with expressions (l-values). Decided to add the attribute *isLVal*. Proof-read one last time.	Just before submitting

5 SDD

Production	Semantic Rules
P → D1 ... Dn	P.*env* = makeEnv(GlobalPreDefs) P.*env* = extendEnv(P.*env*, D1.*fun*) ⋮ P.*env* = extendEnv(P.*env*, Dn.*fun*) D1.*env* = P.*env* ⋮ Dn.*env* = P.*env* **if** ¬existsMain(P.*env*) **then** "error: no main function declared" **if** ¬checkMainRetType(P.*env*) **then** "error: main function must return an int" **if** ¬checkMainArgs(P.*env*) **then** "error: main function can only have parameters of type *char"
D → function T **id**(T1 **id**1, ..., Tn **id**n) Ss	D.*fun* = (T, **id**, ((T1, **id**1), ..., (Tn, **id**n)) Ss.*env* = extendEnv(D.*env*, T, **id**, **fun**(T1 **id**1, ..., Tn **id**n)) Ss.*env* = extendEnv(Ss.*env*, T1, **id**1) ⋮ Ss.*env* = extendEnv(Ss.*env*, Tn, **id**n) Ss.*rets* = T **if** existsEnv(D.*env*, **id**) **then** "error: cannot redeclare function" **if** ¬Ss.*retr* **then** "error: function is not gauranteed to return a value"
Ss → var T1 **id**2 ; Ss3	Ss3.*env* = extendEnv(Ss.*env*, T1, **id**2) Ss3.*rets* = Ss.*rets* Ss.*retr* = Ss3.*retr* **if** existsEnv(Ss.*env*, **id**2) **then** "error: cannot re-declare variable"
Ss → E1 = E2 ; Ss3	E1.*env* = Ss.*env* E2.*env* = Ss.*env* Ss3.*env* = Ss.*env* Ss3.*rets* = Ss.*rets* Ss.*retr* = Ss3.*retr* **if** ¬isEqual(E1.*type*, E2.*type*) **then** "error: can only assign the same type" **if** ¬E1.*isLVal* **then** "error: can only assign to a l-value"

continued on next page

continued from previous page

Production	**Semantic Rules**
Ss → if (E1) Ss2 ; Ss3	E1.*env* = Ss.*env* Ss2.*env* = Ss.*env* Ss3.*env* = Ss.*env* Ss2.*rets* = Ss.*rets* Ss3.*rets* = Ss.*rets* Ss.*retr* = Ss3.*retr* **if** ¬isIntOrPointer(E1.*type*) **then** "error: can only use integer or pointer types for if condition"
Ss → if (E1) Ss2 else Ss3 ; Ss4	E1.*env* = Ss.*env* Ss2.*env* = Ss.*env* Ss3.*env* = Ss.*env* Ss4.*env* = Ss.*env* Ss2.*rets* = Ss.*rets* Ss3.*rets* = Ss.*rets* Ss4.*rets* = Ss.*rets* Ss.*retr* = (Ss2.*retr* ∧ Ss3.*retr*) ∨ Ss4.*retr* **if** ¬isIntOrPointer(E1.*type*) **then** "error: can only use integer or pointer types for if condition"
Ss → while (E1) Ss2 ; Ss3	E1.*env* = Ss.*env* Ss2.*env* = Ss.*env* Ss3.*env* = Ss.*env* Ss2.*rets* = Ss.*rets* Ss3.*rets* = Ss.*rets* Ss.*retr* = Ss3.*retr* **if** ¬isIntOrPointer(E1.*type*) **then** "error: can only use integer or pointer types for while condition"
Ss → return E1 ; Ss2	E1.*env* = Ss.*env* Ss2.*env* = Ss.*env* Ss2.*rets* = Ss.*rets* Ss.*retr* = true **if** ¬isEqual(E1.*type*, Ss.*rets*) **then** "error: wrong return type, must return type" ˆ toString(Ss.*type*)
Ss → { Ss1 } Ss2	Ss1.*env* = Ss.*env* Ss2.*env* = Ss.*env* Ss1.*rets* = Ss.*rets* Ss2.*rets* = Ss.*rets* Ss.*retr* = Ss1.*retr* ∨ Ss2.*retr*
Ss → ϵ	

continued on next page

continued from previous page

Production	**Semantic Rules**
E → **id**1	E.*type* = getVarType(E.*env*, **id**1) E.*isLVal* = true **if** ¬existsEnv(E.*env*, **id**1) **then** “error: variable has not been declared”
E → **str**1	E.*type* = *char E.*isLVal* = false
E → **int**1	E.*type* = int E.*isLVal* = false
E → E0(E1, ..., En)	E0.*env* = E.*env* E1.*env* = E.*env* ⋮ En.*env* = E.*env* E.*type* = getFunType(E.*env*, E0) E.*isLVal* = false **if** ¬existsEnv(E.*env*, E0) **then** “function has not been declared” **if** ¬checkArgTypes(E.*env*, E0, (E1.*type*, ..., En.*type*)) **then** “error: formal and actual types do not match for function call”
E → null(T1)	E.*type* = T1.*type* E.*isLVal* = false
E → sizeof(T1)	E.*type* = int E.*isLVal* = false
E → !E1	E1.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isIntOrPointer(E1.*type*) **then** “error: can only use integer or pointer types for not”
E → -E1	E1.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) **then** “error: can only use integer types for negation”
E → +E1	E1.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) **then** “error: can only use integer types for positive”
E → *E1	E1.*env* = E.*env* **let** T = E1.*type* **in** E.*type* = *T E.*isLVal* = E1.*isLVal* **if** ¬isPointer(E1.*type*) **then** “error: can only use pointer types for dereferencing”

continued on next page

continued from previous page

Production	**Semantic Rules**
E → & E1	E1.*env* = E.*env* E.*isLVal* = false **let** T = E1.*type* **in** E.*type* = *T
E → E1 * E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for multiplication”
E → E1 / E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for division”
E → E1 % E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for modulo”
E → E1 + E2	E1.*env* = E.*env* E2.*env* = E.*env* **if** E1.*type* = int **then** E.*type* = int **if** E1.*type* = *T **then** E.*type* = *T E.*isLVal* = false **if** ¬(isInt(E1.*type*) ∨ isPointer(E1.*type*)) **then** “error: left side can only be integer or pointer types for addition” **if** ¬isInt(E2.*type*) **then** “error: right side can only be integer type for addition”
E → E1 - E2	E1.*env* = E.*env* E2.*env* = E.*env* **if** E1.*type* = int **then** E.*type* = int **if** E1.*type* = *T **then** E.*type* = *T E.*isLVal* = false **if** ¬(isInt(E1.*type*) ∨ isPointer(E1.*type*)) **then** “error: left side can only be integer or pointer types for subtraction” **if** ¬isInt(E2.*type*) **then** “error: right side can only be integer type for subtraction”

continued on next page

continued from previous page

Production	**Semantic Rules**
E → E1 < E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for comparison”
E → E1 > E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for comparison”
E → E1 <= E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for comparison”
E → E1 >= E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isInt(E1.*type*) ∨ ¬isInt(E2.*type*) **then** “error: can only use integer types for comparison”
E → E1 == E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isIntOrPointer(E1.*type*) ∨ ¬isIntOrPointer(E2.*type*) **then** “error: can only use integer or pointer types for comparison” **if** ¬isEqual(E1.*type*, E2.*type*) **then** “error: can only compare the same types”
E → E1 ! = E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isIntOrPointer(E1.*type*) ∨ ¬isIntOrPointer(E2.*type*) **then** “error: can only use integer or pointer types for comparison” **if** ¬isEqual(E1.*type*, E2.*type*) **then** “error: can only compare the same types”

continued on next page

continued from previous page

Production	**Semantic Rules**
E → E1 && E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isIntOrPointer(E1.*type*) ∨ ¬isIntOrPointer(E2.*type*) **then** “error: can only use integer or pointer types for and”
E → E1 || E2	E1.*env* = E.*env* E2.*env* = E.*env* E.*type* = int E.*isLVal* = false **if** ¬isIntOrPointer(E1.*type*) ∨ ¬isIntOrPointer(E2.*type*) **then** “error: can only use integer or pointer types for or”