Project Phase 1B: Symbol Table, semantics, and IR

Total points: 100

Due Date: October 13 2020, 8 am(via Canvas)

Table of Contents

Overview

This part of the project consists of three sub-parts:

- Creating a symbol table
- Running semantic checks
- Generating intermediate code for use in the final phase

You will use the ParseTree Walking mechanisms provided by the ANTLR, using listener and visitor objects. For more information, see Parse Tree Listeners or the ANTLR reference book.

A recommended approach is:

1. Study these mechanisms by going over the relevant parts in ANTLR book and by studying the examples
2. Add code to respective objects to generate the symbol table generation
3. Add code for simple parts of speech such as expression trees consisting of simple operators
4. Add code for the complex expression and parts of speech.

Symbol table

The symbol table is a useful structure generated by the compiler. Semantic checking makes extensive use of the symbol table. The symbol table holds declarative information the constructs that make up a program. The following are the typical entries (this is not an exhaustive list) for a symbol table:

- variable names
- defined constants
- defined types
- procedure and function names and their parameters
- literal constants and strings
- source text labels
- compiler-generated temporaries

You must also manage the attributes for each of the above entries. Typical attributes include:

- textual name
- data type
- dimension information (for aggregates), array bounds
- declaring procedure
- lexical level of declaration (scoping)
- storage class (base address)
- offset in storage
- if record, number of fields and a list of fields with underlying types
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

You have some flexibility in deciding these implementation details, and the above lists are by no means requirements. The guiding principle you should use for building your symbol table and the information you keep in it should be that of utility - store what you will need for the next parts – especially the semantic checking and IR code generation. Your implementation will likely make use of one or more hash-tables to implement a symbol table.

Scoping is a key aspect of symbol tables. By this we mean the following two things:

- The most closely nested rule (that is, references always apply to the most closely nested block). Implement the scoping rules described in the language definition. Additions to the symbol table cannot overwrite previous declarations but instead must mask them. Subsequent lookup operations that you run against the symbol (as you do type checking in part 3, for example) should return the most recently inserted variable definition.
- Declaration before use. A lookup operation on the symbol should never fail. If a lookup fails, then the symbol table has not yet seen a declaration for the reference you are attempting to query. It is a semantic error to use an undeclared value.

The scoping rule we are going to use is the same as used in all block structured languages: inner declaration of a variable name hides the outer declaration. In a given scope, the innermost declaration is the one that is visible and a name used in that scope is bound to that declaration.

As you consider scoping, you are free to implement your symbol table on a scope by scope basis or a global symbol table in which declarations are entered upon entering a scope and deleted upon exiting it. You can use chaining for the entities mapped to the same hash location. The scoping rules are defined for the Tiger language. You should read the relevant symbol table design material from the class text.

A recommended approach is:

1. Read the grammatical and semantic specification of Tiger and decide which values and which of their attributes are going to be held in symbol table.
2. Design a symbol table with hash maps, chains, etc.
3. Implement the symbol table generation

Project requirements

Implement the listener and visitor objects that interface with the parse tree and write tree-walking code to gather and then store the relevant information into the symbol table.

Semantic checking

You implement semantic checks by walking the parse tree using listener and visitor objects.

A recommended approach is:

1. Read the semantic specification of Tiger, which contains the rules about type semantics, including:
- The Operators section discusses types with respect to binary operators.
- The Control Flow section states that the `if`, `while`, and `for` expression headers should all evaluate to true or false.
- The Types section dictates that you enforce a "name type equivalence" for your types
- The other semantic check that must be made is with respect to return statements. A function with no return type cannot have a return statement in it.
2. Complete the binding analysis to determine which variable name binds to which symbol table entry. This is done using look-up mechanism as per the block structure rule. The first step is to decorate the respective nodes of the Parse tree with attributes elicited from the symbol table. The usage of a variable must occur as per its declaration. For example, the number of fields and their usage names of a record must match their declarations for a record.
3. Type-checking is done. Some checks might involve making sure the type and number of parameters of a function match the actual arguments. There are several cases in Tiger where type checking must occur:
- Agreement between binary operands
- Agreement between function return values and the function’s return type
- Agreement between function calls and the function’s parameters

Project requirements

For correct programs, your compiler should pass this part silently and emit nothing. For programs with semantic errors, your compiler should emit an error message stating the problem and a relative place in the Tiger source code.

Intermediate code

The final part of this phase is to convert the program into intermediate code. For the purposes of this project, we will be using 4-address code (or "quadruple" 3-address), which has the following form:

```
op, y, z, x
```

This reads as, "Do operation op to the values y and z, and assign the new value to x." A

simple example of this can be given with the following:

```
2 * a + (b - 3)
```

The IR code for this expression is:

```
sub, b, 3, t1
mult, a, 2, t2
add t1, t2, t3
```

An important characteristic of this representation is the introduction of temporary variables, which the compiler (and therefore you, as the compiler writer) must generate. This also implies that temporary variables must now be made part of the symbol table, and their type should be derived from the result of the operation. For example, in the above expression, `t1`, `t2`, and `t3` would all marked as type `int`. We say that the types for temporary variables *propagate* through the program. You do not have to worry about the number of temporary variables you are creating in the IR. That is, you may assume infinitely many temporary names are at your disposal.

A recommended approach is:

1. Read the lesson on ircodegen (IR code generation).
2. Implement the necessary tree walks and generations of labels and temporaries and
3. Generate the quad IR.

Function calls

The intermediate code has one exception to the 4-address structure. For instructions for function calls, you generate an instruction very similar to that in the source. Function calls with no return values look like this:

```
call, func_name, param1, param2, ..., paramn
```

Function calls with return values will have a similar structure:

```
callr, x, func_name, param1, param2, ..., paramn
```

The bodies of the function calls must be demarcated by adding a suitable annotation (comment):

```
#start_function <function_name> and #end_function <function_name>
```

Requirements

For each correct program submitted to your compiler, the compiler should emit a complete instruction stream of intermediate code for that program. You do not need print any intermediate code for programs with errors.

Turn-in

Grading (100 points)

Deliverables

- Symbol table generation code and symbol table (20 points)
  - The code interfaces to ANTLR generated parser.
  - Implement a symbol table "visualizer" to print out the symbol table.
- Semantic checking code (35 points)
- IR generation code and generated IR (35 points)
- Report (10 points)
  - Design internals.
  - How to build and run the symbol table and IR generation.
  - Test cases. For each test case, include the test case, the symbol table, and the IR.

Format

Please print the symbol table to stdout using the following format:

```
scope 1:
    a, func, int
    b, var, int
    t, type, int
    scope 2:
        c, var, float
        a, func, null
scope 3:
    a, func, float
    c, func, int
```

In this format, the first line will always be the scope number, followed by all the symbols in this scope and its nested symbol tables. Each line will contain *symbolName*, *symbolType*, and *valueType* separated by commas.

In the previous example, scope 1 contains 3 symbols: a, b and t of symbol types func, var and type, respectively, and value types all int. Scope 2 is nested in scope 1 while scope 3 is parallel to scope 1. Function symbols might not have a return value; hence the value type could be null.

If the given Tiger program is both syntactically and semantically correct, then print ”successful compile” to stdout. If there is any semantic error, then your compiler should output an error message that contains the line number and the index of the error within that line. Example:

```
Error : line 6:4: error : unexpected token : var of type VAR
    var temp2 : int := 3; ...
    ^
```

Put the code for part 1A and part 1B together, and provide a single *makefile* to build a single executable file.