Part 3 Extra Credit

CS212

Fall 2005

The Bali language, for which you're writing a compiler this semester, is strongly modeled after the C language, with some ideas from Java. However, it lacks many of the features of C, which make it so powerful. It also lacks certain syntactic elements, that would make writing code easier. To remedy this, here is an extra credit opportunity to enhance the language. You can complete any of the features we've designed below for bonus points in the course. Bouns points are kept separate from grades, and as such, will be considered **after** we compute all numerical and letter course grades. This gives each student/group an opportunity to increase their score without causing others to have lower grades.

In general, if there are multiple subitems for an extra credit, they must be completed in order. For instance, in order to complete (1.2), you should have already completed (1.1)

1 On-heap Objects

1.1 Structures (10 Bonus Points)

Syntax:

program → [globalVars] (function | prototype | structdef) *
structdef → **structdef** name **{** varDecl* **} ;**
type → **struct** name (*****)+
binop → ->
expPart → **sizeof (** type | **struct** name **)**

Semantics:

- Structures are similar to Java objects without any methods - they provide a way for the programmer to group several variables within the same 'object'. Those variables are declared in "Structure" scope, and they are not visible from anywhere, except when accessed with the (->) operator. Like other scopes, the "Structure" scope should not allow duplicate variables. Unlike other scopes, the "Structure" scope should not propagate queries to its parent.
- The syntax above implements structures on the heap. C has structures on the stack, but we will not support those. To restrict our structures, notice that struct types cannot be directly declared, only struct pointers. Furthermore, those pointers should not be dereferencable. For arrays, please note that arrays of structures are not supported, only arrays of structure pointers.
- Structure-based types are valid everywhere in the program, even before the structure is defined. If the structure is never defined, a semantic error should be generated, as opposed to a syntax error for invalid primitive types.
- The struct pointer access operator (->) requires a valid struct pointer on the left side, whose structure has as a field the variable on the right side.
- A struct field is now a valid lvalue (in other words, it can be used on the left side of an assignment). However, it must be enclosed in parentheses like other binary expressions. For example:
  `(a->b) = c;`

- A struct field is now a valid addressable expression (it can be operated upon by **&**). Again, parentheses are required. For example:
  `(c->d) = &(a->b)`
- The **sizeof** builtin function takes as input a type (either primitive or structure), and returns its integer size - the number of memory locations it takes up. Notice that here it is valid to use a struct without a pointer. The typical way this works is:
  `struct str_name* var = <struct str_name*> malloc(sizeof(struct str_name));`
- There is another hidden use of **sizeof**. If you implement structures we expect pointer arithmetic (for example, adding an integer to a pointer), to be translated as follows:

  ```
  type1* a;
  int b;
  ```

  `a + b` translates to `a + b * sizeof(type1)`. This is how pointer addition works in C - when you add 1 to an integer pointer, you are actually adding 4 to the pointer (since the size of an integer is 4 bytes on the x86 architecture)
- **sizeof**, **struct**, and **structdef** are now reserved keywords

1.2 Structs with functions (classes) (10 Bonus Points)

Syntax:

structdef → **structdef** name **{** (varDecl)* **}** [**{** (function | prototype)* **}**]**;**
expPart → **this**

Semantics:

Note: In the following, we use "methods" to refer to functions defined within a struct and "functions" to refer to global functions (those functions defined outside of structs)

- Now structures have an optional block after the variable block which can contain functions and/or prototypes. When parsing these methods, they follow the same rules as functions in a Bali program with respect to defining - a method is considered to be defined when either its function definition or prototype is encountered, and the types, number, and order of arguments and return types of both the prototype (if specified) and function definition must be equal.
- Methods are invoked on an instance of an struct, using the (->) operator. For instance, if there is a structure named **mystruct** with method **foo** that takes a single integer argument, and some other function/method has a variable **x** of type **struct mystruct*** , then **x->foo(3)** is a valid way to call the method **foo** on the struct **x.**
- Each method in a struct has access to an implicit **this** variable whose type is a pointer to the enclosing struct type, which allows access to the variables within the structure that the method was invoked on. For instance, in the example above, the method **foo** has an implicit variable **this** which is equal to **x** (the structure it was invoked on). When compiling methods of a struct, you should add another argument named **this** of a type corresponding to a pointer to the struct (so a method that takes 3 arguments in Bali will actually take 4 arguments when compiled to SaM). When compiling calls to a method on an instance of a struct, pass the reference to that struct in the appropriate argument you have implicitly defined. This allows methods to work on an instance of an object
- **this** is now a reserved keyword.

2 Variable Initializers (10 Bonus Points)

Syntax:

varDeclInit → type name [= expression] (, name [= expression]) * ;
varBlock → { varDeclInit * }

Semantics:

- Initializers contain code that performs a variable assignment immediately after the variable declarations. They can act upon variables in any scope, except structure variables (if structures are implemented).
- The initializer code is to be executed prior to any other statements in the same scope. If multiple initializers are present, they are to be executed in left-to-right, top-to-bottom order.
- A variable is considered declared during the execution of its initializer statement, and any subsequent initializers. It is undeclared for any prior initializers.

  For example:

This is valid, and safe:	`int a=4, b=a;`
This is valid, but unsafe:	`int a, b=a;`
This is valid, but unsafe:	`int a=a;`
This is not valid:	`int a=b, b;`

3 Break/Continue

3.1 Basic break and continue (10 Bonus Points)

Syntax:

statement → **break;**
statement → **continue;**

Semantics:

- **break** immediately exits out of the nearest enclosing **do/until** or **for** loop. It is equivalent to the statement block of a loop completing its execution **and** the conditional expression of the loop evaluating **true** for **do/until** loops or **false** for **for** loops. After completion of the **break** statement, execution should continue immediately after the immediately enclosing loop.
- **continue** immediately jumps to the end of the current iteration of the nearest enclosing loop. It is equivalent to the statement block of a loop completing its execution. For a **do/until** loop, this means that the conditional is evaluated, and a new iteration starts if it is **false**. For a **for** loop, the third expression is executed, then the second is evaluated and a new iteration starts if it is **true**.
- Both statements are valid inside loops only. Any use outside of an enclosing loop should raise a semantic error.
- **break** and **continue** are now reserved keywords.

3.2 Break to label (10 Bonus Points)

Syntax:

statement → **break** [name]**;**
statement → **continue** [name]**;**
statement → [**@**name] **do** statement **until (**expression**);**
statement → [**@**name] **for(**[expression]**;**[expression]**;**[expression]**)** statement

Semantics:

- Loop structures now can have an optional label in front of them. This label begins with an **@** symbol but is otherwise the same as a variable identifier.
- Both **break** and **continue** now can accept an optional label argument. When a label is specified, their effect is now applied to the appropriately labeled enclosing loop. For instance,

```
@loop1 for (x = 1; x < 10; x = x+1) {
      @loop2 for (y = 1; y < 10; y = x + y) {
            @loop3 do {
                  if (z == 3) break loop1;
                  z = z + x*y;
            } until (z > 10);
      }
}
```

  will break out of all three loops when z is equal to 3.
- **break** and **continue** should only accept labels that refer to enclosing loops in the current function. It is a semantic error for a label to be specified that refers to a non-enclosing loop, or a loop in another function.
- If nested loops have the same label, **break** and **continue** when applied to that label refer to the innermost enclosing loop with that label.