Javalette compiler (JLC)

Joakim Öhman

04-05-2014

1 Running the compiler

JLC needs to first be compiled before it can run. To compile it, run the 'make' command inside the 'src' directory. One can then run './jlc [-O] <SourceFile>' from the root directory of the JLC package.

JLC generates LLVM code, by first parsing and type checking, and then generating LLVM code. If everything goes right, the source file 'program.jl' will compile to 'program.ll' and then run the LLVM assembler to generate 'program.bc'. These generated files will be created in the same directory as 'program.jl'. It will also generate 'a.out.bc' which is the code linked with the runtime library and an executable file 'a.out' for running the program. These are created in the current working directory. The compiler will also output 'OK' to standard error.

If it fails to parse the souce code, it will output an syntax error with an explantion and if it fails to type check the source, it will output an type error with explantion. JLC will also output 'ERROR' to signal the test suite that compilaton failed.

The only flag JLC has is '-O', which runs the LLVM optimization program for the generated code.

2 Javalette language

For the language specification, we refer to the 'Javalette.cf' file, the BNF converter source file for the language. It is located in the 'src' directory.

2.1 Shift/reduce conflicts

Our grammar for the language has one shift/reduce conflict. This conflict is the dangling else problem, which is a common problem in C-like languages. This conflict occurs since 'else' is optional in an 'if' statement. Since it is optional, the LR parser must try both the 'if' statement and the 'if-else' statement while parsing. This leads to ambiguous parsing of certain programs.

3 Makefile instructions

The makefile has a couple of useful instructions that one can use. We will here mention them and what their actions are:

- all - Compiles the JLC source code and assembles the runtime library. This is needed since some different LLVM versions have differentiating bytecode formats.
- clean - Removes all files generated at compile time, including BNFC generated files.
- clean-nobnfc - Removes all files generated except those generated by BNFC. This might be useful if one does not have BNFC installed or want to distribute to people without BNFC.

4 Implemented extensions

We will here go through the extensions that is implemented and some details about them. The implemented extensions are:

- One-dimensional arrays and for loops
- Multidimensional arrays
- Dynamic data structures
- Object-orientation
- Object orientation with dynamic dispatch

This should add up to 5 credits. Now we will go through the specific features and implementation details of each extension.

4.1 One-dimensional arrays and for loops

When an array is allocated, it calculates the memory space that is needed for that array at runtime, since the size parameter can be a variable. The allocation code is inlined.

For-loops use an internal variable to keep track at what element it is at in the array. The array used in a for-loop and its size is loaded once before the loop body. The internal variable is the incremented and it keeps running the loop body until the variable is equal to the array size.

4.2 Multidimensional arrays

Multidimensional arrays have a different construction procedure compared to one-dimensional arrays. The construction procedure of multidimensional arrays is an internal LLVM function. The function, called multiArray in the runtime library, recursively allocates each subarray following an internal array to keep track of each array size. One-dimensional arrays and multidimensional array do not use the same constructor since the multidimensional one must generate and keep track of more data in the construction process than the one-dimensional.

For-loop can be used with multidimensional arrays, when you do you get the inner array at the corresponding position assigned to the iterator variable.

4.3 Dynamic data structures

The delimitation for this extension is that typedef can only be used for structure types. During type checking, all type definitions are translated into their corresponding structure pointer type.

4.4 Object-orientation

The delimitation for this extension is that a class extends an other, it must be defined before the extending one. This is to prohibit circular dependencies.

Objects are stored like structures. During compilation class variables operations are translated to structure operations.

All class methods' first argument is the object that is calling it.

Here we will explain how inheritance is implemented. Let A and B be classes, B extends A and an object of B. when using in cases where you use objects of type A, is being cast to type A. Note however that casts are otherwise not supported and can never be explicit. This implementation relies on the ordering of the class variables is consistent, which is ensured in the class type preparation during compilation.

4.5 Object orientation with dynamic dispatch

The first element of all object structures is a method table. The table is an LLVM array with the methods stored as function pointers. Each method's number is saved during compilation for future lookup during method calls.

Each class creates a constructor for itself, where objects are being allocated and the method table is being instantiated.