CS4212 Project: Jlite Compiler

I. Parsing of JLite

Parsing of JLite consist of two steps, tokenizing and parsing. Tokenizing is done using JFlex and parsing is done using CUP.

I.1 Tokenizing

JLite string is tokenized into series of tokens that are represented by an enum in Java. The enum contains all of the special character such as brackets, operators, etc. It also contains some reserve words such as “true”, “false”, etc. However, the word “main” is not included since it is not a reserved word, but a specification for the main class. For that reason, it will be handled separately from the tokenizer.

I.2 Parsing

Parsing is done by specifiying the Jlite grammar to CUP LALR library. Many grammar specified in the project description can be ported directly into CUP grammar. However, some of the grammar have to be changed to fit LALR parser. One such grammar is the expression grammar. In the project description, boolean expression and arithmetic expression is splited into two different part. To simplify the parsing, boolean and arithmetic expression is considered the same with different operation precedence. Thus resulting in a boolean and boolean arithmetic operation. However, this should not be a problem since it will be handled in typechecking phase.

During this stage, it also ensure that the first class satisfy the requirement to be a main class. This have to be done separatly from the parsing process because the parser does not differentiate the main class and the other classes. It is done this way to simplify the parsing process because the main class is similar with the other class but with some restriction. It is easier to check for this restriction later than having different grammar for the main class.

The result of parsing phase is a JLite abstract syntax tree (ast). Each ast node represent an opearation of JLite such as assignment, function call, or an Atom. A statement is defined to be a statement in JLite. An atom is a unit that can be evaluated and returns a value. For example, “a + b” is an Atom because of we know the value of “a” and “b”, it can be evaluated and returns a value. Meanwhile, “x = a + b” is not an atom since it does not returns anything, thus classifiying it as a statement. This design also makes function call both a statement and an atom since it can be called without storing the return value or use the return value for other operation. In this stage, call statement and call atom is not differentiated yet since the ast does not need to know what the function call actually do.

Initially, the parser is expected to fix simple errors such as missing semicolon. However, it was not done due to complication of adding it later in the development and time constrant.

II. Type Check

Type checking is done in the JLite ast. All static check is done on function level. The checking process is as follows:

- Global environment setup
- Local environment setup
- Code walkthrough

II.1 Global Environment Setup

Global environment setup sets all global reference that can be accessed from all function. In JLite the only global reference is the class descriptor. A class descriptor consist of fields and function that a class have. This descriptor helps when the checker need to check a type or existence of a field or function. In the class descriptor, function is treated similarly with fields. It generates a type for each function, the type comprised of the return value and the type of its parameters. The assignment of type allows the posibility of overloading functions because the checker can request the class descrptor to return a function with a specific arguments. Unfortunately, this feature is not implemented

II.2 Local environment setup

Local environment includes the functions' parent class field, arguments, and local variables. This environment is useful to find the reference and type of all variable used in the function. In this implementation, it is possible to overload class variable with arguments such that the only way to access the class variable is by using "this" keyword.

II.3 Code walkthrough

This is the stage where the type checking actually happen. There are several cases of checking:

- Return
  return variable's type should match the function return type. Empty return is evaluated to return a "Void" type.
- Assignment
  The type of left and right side of the assignment must be equal.
- Dot operator
  The left side of the dot must be a class and it also have to have the field/function with the same name of the right side. Then, return the type of that field.
- Function call
  The callee must have a "Function" type and the arguments types must match its requirement.
- Boolean/arithmetic operation
  Each operator have a requirement for the operand and and its return value.

The type checking will try to collect all of the error before returning. Thus it will catch all type errors.

III. Intermediate Code Generation

The intermediate code that generated in this stage is called IR3 that also have the properties of three address code. It is generated based on the ast with correct typing to simplify the generation procecss. The IR3Builder class is used to wrap the code generation processes. The IR3builder also provides all generator function that can be used to generate label and identifier.

The implemented IR3 will also generate one-time used identifier for intermediate computation such as long expression. The IR3 generated also follows similar pattern to the ast, that it is also a tree. In this representation, function call for statement and atom is differentiated because the assembly code that they generate will be different. The major syntax changes from ast to IR3 primarily are while and if.

IV. Register Allocation

Register allocation is done using "latest next used variable" heuristic. If there is no free register, this heuristic will choose to spill a variable that is either no longer used in the execution or a variable that the next usage is the farthest. The next variable usage can be computed using dfs for each variable. However, doing the traversal blindly can result in high computational complexity. To optimize this, the traversal can be halted whetever it found a usage of the variable.

To simplify the registry allocation, a concept of “prerequisite register state” is introduced. The concept applies on block level and it specifies that all program path that will enter a block must must have their register match the prerequisite state (what variable in what register). This concept makes it eaiser to allocate register and a variable can be assigned to different register and share same register depends on what pregram point the register state is in. This approach can be optimized by removing unused register from the prerequisite if such variable is not used in a block. The allocation can also better computed using dynamic programming approach to select which register can be spilled.

After registers are allocated, the register allocation is manifested in IR3 code. This approach extends the IR3 capability to also specify which variable must be loaded or stored from register. This is an example of the extended IR3:

Store{varName='d', register=1}
Load{varName='help', register=1, setOnly=true}

In the example, the IR3 code instruct the asm generator to store variable ‘d’ from register 1 to stack memory. The other instruction tells the generator to load variable ‘help’ from memory to register 1. the ‘setOnly’ option is true to tell the generator that the variable ‘help’ is does not need any data from the memory. So, it only tells the generator to not write a load statement and only mark register 1 to contain variable ‘help’. The ‘setOnly’ option is not filled by the allocator, but by optimizator later. In cases where the variable from a register is no longer used, the allocator will omit the ‘store’ instruction.

V. Optimization

There are several optimization included in this project:

- Dead Code Remover
- Load Remover
- Constant Folder
- Unused Variable Remover

V.1. Dead Code Remover

The intention of this optimization is to remove unreachable statement from the code. This optimization can be done before and after the register allocation. This optmiziation is done by traversing the code and removing unvisited statement.

V.2. Load Remover

The intention of this optimization is to remove unnecessary load instruction produced by register allocator. This optimization is done by removing load statement that is not used, such as overwritten immediately. This can be done by traversing the graph for each load statement and try to find any usage of such variable. If no such statement found, the ‘setOnly’ flag of the load statement will be set to true.

V.3. Constant Folder

Constant folder done to precompute all variable that the value is already known such as

```
a = 3;
b = 4;
c = a + b;
```

the last statement can be optimized to “c = 7;” this optimization is done by constructing a topological sort from the variable assignment. However, this optiomization will be skiped if any of the operand includes a class field

because of the volatile nature of it. Based on the topological order of the assignment, it tries to compute the value of each variable. If it found that a variable value is constant (same across all assignment), it will mark it as a constant and use its value to compute other assignment. Howerver, this optimization produces a lot of unused variable.

V.4. Unused Variable Remover

This optimization is intented to remove unused variable produced by constant folder and reduce the stack memory usage. Criteria of a variable that will be removed is that it is not a class and it is never used or stored back to the stack memory. This optimization is implemented similar to the load remover.

VI. Assembly generation

The assembly generated by the generator is a direct translation of the IR3 code.The AssemblyBuilder class is a wrapper to generate the assembly code. It also have a register manager that keeps track on which variable in which location. The stack manager is a class that keeps track on the variable type and offset for each variable and fields.