Lim Yan Peng, Gary
A0189806M

AY2021/2022 SEM 1

Programming Assignment 3 – Report

1 Overview

The report describes the implementation details for a compiler that parses JLite, a simplified version of Java, into ARM assembly code. The compiler consists of three components. The front end, the middle end and the back end. JLite code is passed in from the front end and eventually processed into ARM code returned by the backend. Instructions on how to run the code is attached in section 6 of the report.

2 Front End

The front end has two components. The Lexer is responsible for lexical analysis where every character in the input file is grouped into tokens based on terminals found in the context free grammar. The Parser performs syntactic analysis and use the tokens produced by the Lexer to match against available production rules in the context free grammar to form a parse tree.

2.1 Lexer

The Lexer is design is based on a finite state machine. The default state is START. Depending on what characters are seen next, the state changes and characters are consumed to form tokens. This continues until a specific end character is encountered, whereby the state goes back to START again. The types of characters are classified as follows:

Category Name	**List of symbols**	**Remarks**
NAME_START	[A-Za-z]	Start of a name (cannot include underscore)
NAME_BODY	[A-Za-z_0-9]	Body of a name (can include underscore and digits)
DIGIT	[0-9]	Digits
WHITESPACE	[' ', '\t', '\n', '\r', '\v', '\f']	Represents whitespace character
MONO_LITERAL	['{', '}', ';', '(', ')', ',', '.', '+', '-', '*']	Represents a single-character token
SLASH	'/'	Mark the start of comments / division symbol
COMP_LITERAL	['=','<','>','!']	
AND	'&'	
BAR	'|'	
DOUBLE_QUOTE	'\"'	

The states of the FSM are listed below. It also explains how the state is reached.

STATE_START - default
STATE_NAME - when NAME_START is encountered in STATE_START
STATE_DIGIT - when DIGIT is encountered in STATE_START
STATE_COMP - when COMP_LITERAL is encountered in STATE_START
STATE_BAR - when BAR is encountered in STATE_START
STATE_AND - when AND is encountered in STATE_START
STATE_STRING - when DOUBLE_QUOTE is encountered in STATE_START
STATE_STRING_ESCAPE - when '\' is encountered in STATE_STRING
STATE_SINGLE_SLASH - when SLASH is encountered in STATE_START
STATE_COMMENT - when SLASH or * is encountered in STATE_SINGLE_SLASH

Each of these states may transit to other states depending on what characters are being consumed next. The order of checks goes from left to right as some of the categories may be superset of others (e.g. NAME_BODY is a superset of NAME_START). As some tokens may consists of multiple characters, their characters are usually place in a buffer until a state transition back to START triggers the Lexer to flush all characters out and concatenate them as a single token.

A ‘x’ symbol represents it is an unexpected token at that state and an error should be thrown.
A ‘-‘ symbol represents the character category is not checked for that state (because it is probably a subset of another category)
A bolded **START** represents that the collected characters in the buffer are flushed to become a single token upon reaching that state (acceptance state).

States	**Character Category**										
	NAME_START	DIGIT	NAME_BODY	WHITESPACE	MONO_LITERAL	SLASH	COMP_LITERAL	AND	BAR	DOUBLE_QUOTE	BACK_SLASH
START	NAME	DIGIT	x	START	**START**	SINGLE_SLASH	COMP	AND	BAR	STRING	x
NAME	-	-	NAME	**START**	**START**	**START**	**START**	**START**	**START**	**START**	**START**
DIGIT	**START**	DIGIT	x	**START**	**START**	**START**	**START**	**START**	**START**	**START**	**START**
COMP	**START**	**START**	**START**	**START**	**START**	**START**	If ‘=’, add to buffer **START**	**START**	**START**	**START**	**START**
AND	x	x	x	x	x	x	x	**START**	x	x	X
BAR	x	x	x	x	x	x	x	x	**START**	x	X
STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING	**START**	STRING_ESCAPE
STRING_ESCAPE	STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING	STRING
SINGLE_SLASH	**START**	**START**	**START**	**START**	If ‘*’, COMMENT	COMMENT	**START**	**START**	**START**	**START**	**START**
COMMENT	COMMENT	COMMENT	COMMENT	If \n and only has single-line comment START	If ‘*’ and next is ‘/’, and has multi-line comment START else, COMMENT	If next is ‘*’, add to multi-line comment stack COMMENT	COMMENT	COMMENT	COMMENT	COMMENT	COMMENT

Note:

a. In STATE_COMP, the buffer at that point contains only a single character that is one of ['=','<','>','!']. It will only concatenate with the next symbol if it is ‘=’, otherwise, it flushes the single relational symbol / equal sign
b. The only difference between STATE_STRING and STATE_STRING_ESCAPE is that the latter do not terminate the string upon seeing DOUBLE_QUOTE
c. For STATE_SINGLE_SLASH, it will only transit to STATE_COMMENT if ‘/’ or ‘*’ are encountered, otherwise, the first ‘/’ will be treated as a division symbol and be flushed as a token
d. For comments, two different additional variables are maintained to track the sub-states
    a. BOOLEAN: TRUE if it is currently in the middle of a single-line comment

    b. STACK: Contains varying number of /* that needs to be matched with */, similar to how bracket matching algorithm works
e. For STATE_COMMENT, no situation will cause it to flush since we do not keep comment body as tokens

The output is a list of tokens of all recognizable terminal symbols with the comments stripped away. This list will be passed to the Parser to be matched against production rules of the given grammer.

2.2 Parser

The parser accomplishes its tasks by advancing tokens one by one (and occasionally performing lookaheads) to match symbols described in the JLite grammar. It starts off by parsing the Program, within which it will attempt to parse the MainClass, ClassDecl and so on.

The parser contains 3 main types of functions in its API:

accept(expected) – checks whether the next token matches the **expected** string. If it matches, it is accepted and the function returns TRUE. Else it is not accepted, and returns FALSE. No error is returned, so that the code calling it can check if any other symbols are accepted. Used for solving ambiguities

expect(expected) – checks whether the next token matches the expected string. If it matches, consume it and advance to the next token. Else, throw an ERROR

parseXXX() – A broader function that expects a certain symbol. E.g. **ParseClassDecl()** parses a *ClassDecl* symbol from the grammar by calling at specific junctures other **parseXXX()** functions, **expect(expected)** to consume terminals and **accept(expected)** to check whether a particular terminal can be accepted.

The logic for parsing symbol XXX is contained within the function parseXXX. This function will expect terminals or parse symbols in a specific order according to JLite specifications. For example, parsing the symbol VarDecl would happen in 3 steps:

```
parseType()
parseId()
expect(;)
```

Within parseType(), it would expect a capitalized string while within parseId(), it would expect a non-capitalized string.

Ambiguities

There are a few cases of ambiguities in the JLite grammar. This is first addressed by doing left-factoring or removal of left-recursion in the grammar whenever possible. But this will not solve all ambiguities. The following highlights some strategies that are used in the parser:

Left Recursion Removal

For Atom:
<Atom> → ***<Atom>*** . *<id>* | ***<Atom>*** (*<ExpList>*) | this | *<id>* | new *<cname>* ( ) | (*<Exp>*) | null

<Atom> → this *<Atom'>* | *<id>* *<Atom'>* | new *<cname>* ( ) *<Atom'>*| (*<Exp>*) *<Atom'>*| null *<Atom'>*
<Atom'> → . *<id>* Atom' | (*<ExpList>*) *<Atom'>* | e

For SExp:
<SExp> → ***<SExp>*** + *<SExp>* | STRING_LITERAL | *<Atom>*

<SExp> → STRING_LITERAL *<SExp'>* | *<Atom>* *<SExp'>*
<SExp'> → + *<SExp>* *<SExp'>*

The strategy used for SExp is similarly applied to AExp, BExp, Term and Conj so they will not be repeated again. After left-recursion removal is done, we can easily perform lookahead to see if the symbol has terminated.

Lookahead
<ClassDecl> --> class <cname> { <VarDecl>* <MdDecl>+ }
<VarDecl> --> <Type> <Id> ;
<MdDecl> --> <Type> <Id> (<FmlList>) <MdBody>
...
In the above example, when parsing ClassDecl, we do not know where is the end of the VarDecl list since it shares the same prefix as MdDecl. However, a notable difference between VarDecl and MdDecl is that VarDecl expects a ; in the 3rd position. We can thus lookahead by 2 symbols to see whether it is the end of the VarDecl list.

Backtracking
In resolving expressions, it is difficult to determine from the first symbol which of the three expression types it could be (String Expression, Arithmetic Expression, Boolean Expression), especially if the first symbol is an atom, since all three expression types could begin with atom.

// String
atom1 + atom2 + atom3 + STRING_LITERAL

// Arithmetic
atom1 + atom2 + atom3 + 2

// Boolean
atom1 + atom2 + atom3 + 2 >= 0

In case 1, we can only confirm that it is a String Exp only after we see the STRING_LITERAL.

In case 2, we might wrongly assume that it is a String expression until we see the first INTEGER_LITERAL. In that case, we have to backtrack and reparse the atoms as a Arithmetic Expression.

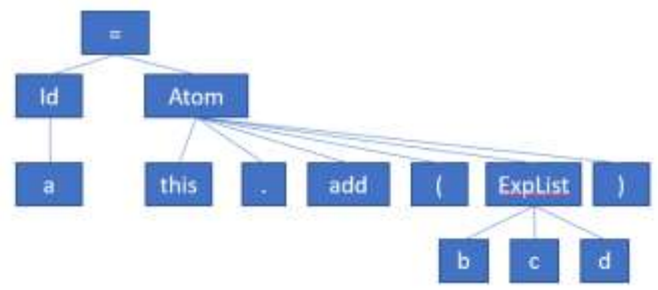
In case 3, we might wrongly assume that the whole expression is an Arithmetic Expression until we see the binary operator. In that case, we have to backtrack and reparse the atoms as a Boolean Expression that has a Relational Expression as its first part.

The solution behind this is to backtrack (by resetting the current index of the token list to the start of the expression) whenever a terminal suggests a different expression type. The algorithm for parsing

Expressions make use of a 3-bit number to track the current state. Each bit represents whether the corresponding state is still possible or not - (BOOL, ARITH, STRING). For instance, if a number token "42" is seen or a minus, multiply or divide token is seen, then the 3rd bit will be set to zero and the expression will not attempt to look for string literals. Conversely, if a string token like ""Hello World"" is seen, then the BOOL and ARITH bit will be set to zero. This helps to ensure only valid expressions are accepted without taking into account the types of atoms yet.

The output of a parse is a Parse Tree built from the PNode class defined in parser.py.
PNode:
self.value:str – the terminal symbol it represents
self.children:list – a list of string or other PNodes representing their structure in a parse tree

For example, a statement "a = this.add(b, c, d)" would have a parse tree that looks like this.



The parse tree do not have any other purpose apart from pretty printing. It has to be further processed in the Middle End into an Abstract Syntax Tree (AST).

3 Middle End

3.1 Name Checking

Convert parse tree to AST

The output of the parser actually generates a parse tree that contains a lot of unnecessary symbols like ;. The parse tree thus has to be converted into a proper AST. This step was surprisingly laborious, especially since the children of each parse tree nodes are composed of mixtures of other nodes and strings, which means a lot of code logic is required to verify the type of the children. The step here could have been avoided had the AST been constricted directly from part 1 of the assignment instead of the intermediate parse tree that had aided in the pretty printing.

Name Checking

First, I check for existence of duplicate class names by storing each class name as a key in a dictionary.

After which, I adopted the same method for checking the existence of duplicate class fields and method variables. Basically, each field / variable name and its type is stored as a key-value pair in a dictionary.

The checking of duplicate method declaration is slightly more complex as my compiler allows method overloading. To go about distinguishing methods with the same name but different sequence of parameter types, a two-level dictionary was used. The set of all possible methods for a class is stored in a dictionary that maps method names to another dictionary, where the inner dictionary maps a tuple of strings (representing the method's parameter types) to the return type of the method. This allows methods of the same name but different sequence of argument types (e.g. a(Int x, Bool y) and a(Int x, String y)) to be distinguished by the inner dictionary. For instance, the two methods (assuming they belong to classA and have return types of Bool and String respectively) will be stored in the dictionary as such:

classA
- a
    - (Int, Bool)
        - Return type: Bool
    - (Int, String)
        - Return type: String

3.2 Type Checking

This part is implemented based on Assignment specifications. A TypeEnvironment object is prepared before the start of the type checking for each method. This object maps classnames to the dictionary of methods and the set of fields (and their types). The local variables (actual and formal variables) of a method is also stored within the TypeEnvironment. This allows method types, return types and expression types to be easily verified when traversing down the AST.

Most expressions and statements have an expected type associated with it. For example, the type of an add expression can either be a String or an Int expression. To further resolve this, it will inherit the type of the left expression. The right expression will subsequently need to match the type of the add expression. The type of a Call statement will depend on the return type of the method being called. Some other checks include:

- type of expressions of return statements match the return type of the method
- last statements of each method match the return type of the method
- last statements of then and else block of if statements match each other

For call statements, the calling class must be resolved first. It can either be the '**this**' class for a local call or a class represented by some expressions. Subsequently, the types of each argument expression are also resolved. Only then can we check that the class contains a method with the respective parameters.

For field access or variable access, the concept is similar to calls. For variables, if it does not belong in the scope of the method, it will check the type environment to look for the variable id from the fields of the current class and classify it as a field access if found. Field access are directly checked against the type environment to verify if the class has a field with the particular id and the required type.

Null literals will take on the expected type of the expression it is used in. If it is used as a return expression, Null will take on the return type of the method where the return statement is found, provided that the type is Nullable (includes all defined classes and String).

Intermediate Code Generator
From the AST, every AST statement node will be resolved into a sequence of IR3 statements that will be concatenated together based on the order of the AST statement in the block. Every AST expression node will be resolved into a sequence of IR3 statements as well as an identifier, which can be a temporarily initialized variable. Children nodes will pass on their generated IR3 statenebts to the parent nodes to be collapsed into a larger list of IR3 statements.

The translation scheme for each node to the corresponding IR3 can be understood easily from the code so only a few notable details will be covered. Firstly, all method names are changed to the concatenation of the classname and the order of the method's declaration in the class. This is so that overloaded methods can be identified more easily.

Temporary variables with the pattern '_eXX' is also introduced to hold the values for intermediate computation. The XX represents an auto-incrementing counter to prevent overlaps of variable names. The counter is reset before processing each Method Declaration AST node since the scope for every method will never be overlapping.

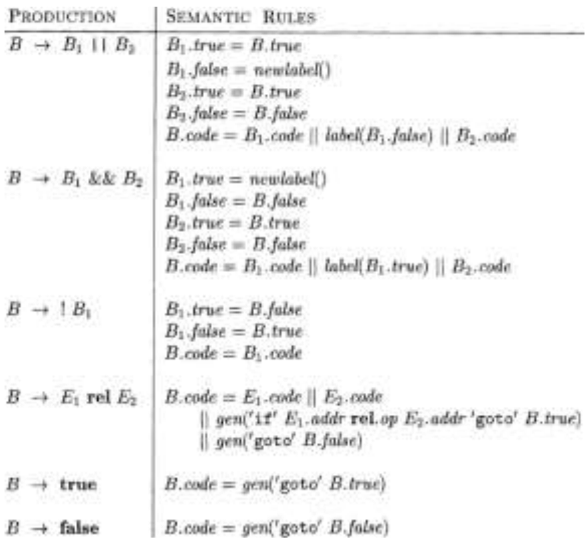
Similarly, labels are numbered starting from 0 onwards and auto-incremented whenever a new label is introduced. For labels however, their numbers are not reset so that each label number is unique in the entire program, facilitating its usage in the ARM assembly generator.

Boolean expressions are handled the same manner regardless of whether they appear in a 'If' statement or 'While' statement or an assignment statement. In the AST, all boolean expressions are formed from a BoolOrNode, which consists of a list of one or more BoolAndNode that are joined as a disjunction expression. BoolAndNode represents a conjunction of BoolRelNodes. BoolRelNodes can be an actual relational expression or an entity from the <BGrd> entity described in assignment 1, which is either 'true', 'false', Atom, or !<BGrd>.

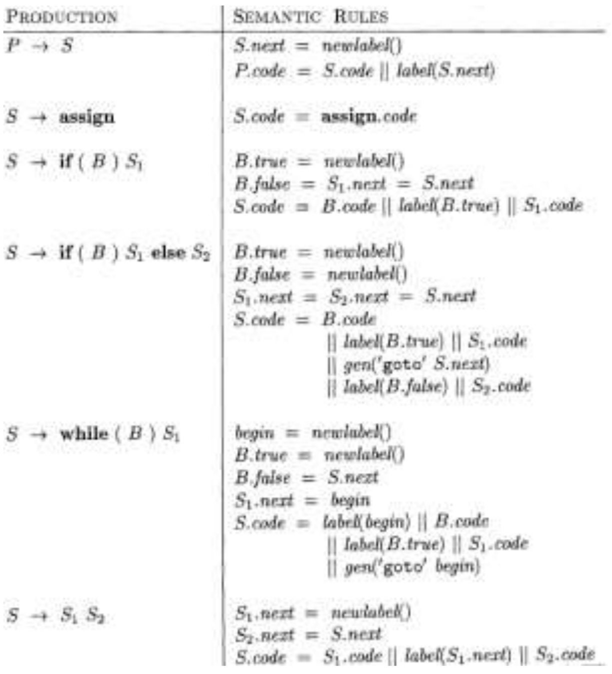
BoolOrNode
- BoolAndNode1 || BoolAndNode2 || BoolAndNode3 …

BoolAndNode
- BoolRelNode1 && BoolRelNode2 && BoolRelNode3 …

The key strategy used in constructing the IR3 is to make use of the disjunction and conjunctions to include short circuiting (goto-label combination) whenever necessary. The logic is derived from the lecture slides and textbook definitions as shown below.



PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mathbin{\|\|} B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mathbin{\|\|} label(B_1.false) \mathbin{\|\|} B_2.code$
$B \rightarrow B_1 \mathbin{\&\&} B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mathbin{\|\|} label(B_1.true) \mathbin{\|\|} B_2.code$
$B \rightarrow \,!\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mathbin{\|\|} E_2.code$ $\mathbin{\|\|} gen($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto' $B.true)$ $\mathbin{\|\|} gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'goto' $B.false)$



PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mathbin{\|\|} label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**.$code$
$S \rightarrow$ **if** $(\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mathbin{\|\|} label(B.true) \mathbin{\|\|} S_1.code$
$S \rightarrow$ **if** $(\ B\ )\ S_1$ **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mathbin{\|\|} label(B.true) \mathbin{\|\|} S_1.code$ $\mathbin{\|\|} gen($'goto' $S.next)$ $\mathbin{\|\|} label(B.false) \mathbin{\|\|} S_2.code$
$S \rightarrow$ **while** $(\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mathbin{\|\|} B.code$ $\mathbin{\|\|} label(B.true) \mathbin{\|\|} S_1.code$ $\mathbin{\|\|} gen($'goto' $begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mathbin{\|\|} label(S_1.next) \mathbin{\|\|} S_2.code$

Middle End Optimizations

Optimizations are also included at the end of the IR3 generation. These optimizations are deemed necessary, otherwise the IR3 program would become unnecessarily long due to the implementation used. Hence, no options are provided to disable this optimization. You may see it as a form of trimming. Some optimization methods adopted are:

- Eliminating unreachable code
  - From 'goto' / 'return' statements to the next 'label' / end of code
- Flow-of-control optimizations (multiple consecutive jumps)

Initial	Eliminating consecutive jumps	Eliminating unused labels

Label 1:
goto 2;
Label 2:
goto 3;
Label 3:
goto 4;
	Label 1:
	goto 4;
	Label 2:
	goto 4;
	Label 3:
	goto 4;
		Label 1:
		goto 4;

- Collapse adjacent labels
- Collapse if-goto and goto statements if they are adjacent and goto the same label
- Collapse goto and label statements if they are adjacent and refers to the same label

Initial	Collapse if-goto and goto	Collapse goto and Label	Collapse adjacent Label

if (_e1 == _e2) goto 2;
goto 2;
Label 2:
Label 3:
	goto 2;
	Label 2:
	Label 3:
		Label 2:
		Label 3:
			Label 2:

The optimization is repeated until there is no more change in the length of the IR3 statement list.

4 Back End

The back end consists of three steps. The first step is the construction of the Control Flow Graph (CFG). The second step is register allocation for each method body. This step also calculates the number of additional stack space a call frame requires for its local variables or from spilling. The third step makes use of the information generated in the register allocation step to convert all IR3 statements into ARM assembly code.

4.1 Control Flow Graph Construction

Each method body will include a single list of IR3 statements. The IR3 statements are grouped into blocks based on the control flow path. For instance, each label statement indicates the start of a new block as control can enter this point from not just the preceding statement. Each goto statement and each if-goto statement represents the end of a block. After blocks are constructed, their list of parents and list of children are also generated based on the goto relationships and the ordering of statements.

Eventually, the CFG will consist of a root basic block, a leaf basic block (the final one) and a mapping of block names to all the blocks. Each block will have its own list of IR3 statements, as well as the lists of parent and child blocks.

4.2 Register Allocation

Compute Variable Live Range

Before registers can be mapped to variables, the live range of each variable must be known so as to deconflict register usage. This is done in a breadth first search approach from the bottom (leaf block) of the CFG. all blocks are processed to obtain the variables used per line (UsePerLine) and the variables defined at each line (DefPerLine). The mapping LivePerLine indicates variables that should be live from that point onwards because they have been defined and will be eventually used. The live-in and live-out for each block is also computed here.

1. The live-out is initialized to be the empty set
2. For each block (starting with the leaf block)
    a. The live-out of the block is the union of the live-in of all the children blocks
    b. Iterate upwards from the **last** IR3 statement to the **first**
    c. Compute the UsePerLine and the DefPerLine based on the IR3 statement type. E.g. all arguments of a Call statement are used. All left hand side variables of assignment statements are defined.
    d. For each variable in the DefPerLine, remove the variable from the LivePerLine mapping
    e. For each variable in the UsePerLine, update mapping of the variable in the LivePerLine to be the current line
    f. After the first statement has been processed, the live-in for the block will be the LivePerLine set.
    g. Place all the parents of this block on the DFS stack to visit next
    h. Mark this block as visited to prevent infinite loop

Allocate Registers based on Live Range Info

Registers are allocated using a naïve register allocation approach that assigns any free register to a variable that needs it. It tries to minimize register hogging by freeing up registers for variables that are no longer live (will not be used at any point later in the code).

The following shows the step performed for creating the register to variable (regToVar) mapping at each line of a block, starting from line 0:

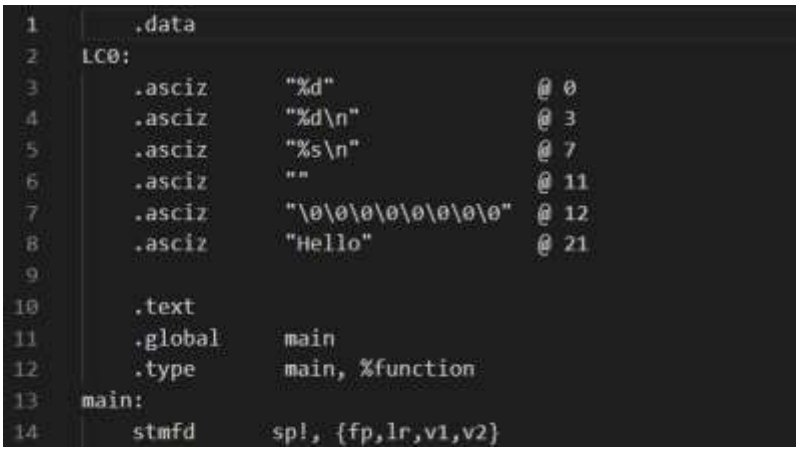
1. When progressing from line i-1 to line i, if any variable went from live to non-live, free the register right away.
    a. For line 0, the previous set of live variables will be the block's live-in
2. If the line involves a function call (printf, scanf, malloc, local call)
    a. Free a1 to aN registers depending on the number of arguments needed.
        i. If these registers are previously mapped to variables, either assign them to other registers, or spill them to memory if no registers are available
    b. Map a1 to aN registers to the variables or values they are supposed to have (arg_0 to arg_N)
3. ToAllocate = Union(LivePerLine, UsePerLine)
    a. For each variable in ToAllocate, if variable has no register allocated to it previously AND is *used or defined in the current line*, assign the variable a register
    b. The above also implicitly delays the assigning of registers to variables that do not need to be used yet
4. When assigning new register
    a. If there is free register, use it, and update accordingly
    b. If there is no free register, spill
5. Who to spill
    a. Choose the one with the latest next use. Spill this variable and assign it a specific position on the stack. It will be referenced later with a specific offset from the stack pointer
6. Carry forward the regToVar mapping to process the next line (or next children block)

4.3 Instruction Generation

Pre-processing

This step scans all the string literals used in the program and declare them (as well as their relative offset from the .data label) in the data segment. If they need to be used, their memory location is retrieved by their offset. All format specifiers ("%d" for reading integer with scanf, "%d\n" and "%s\n" for printing integers and strings respectively with printf) are also stored in the data to enable easier calling of scanf and printf which uses these format specifiers as their first argument. A null string "" is also stored in the data segment. All initialized string vars and fields will take on this value if unspecified. The data segment also has a sequence of null characters "\0\0\0\0\0\0\0\0" that is used as the write space for scanf so that integers can be read from that address into registers subsequently.

E.g. Where "Hello" is a string literal used in the program



```
    .data
LC0:
    .asciz      "%d"                @ 0
    .asciz      "%d\n"              @ 3
    .asciz      "%s\n"              @ 7
    .asciz      ""                  @ 11
    .asciz      "\0\0\0\0\0\0\0\0"  @ 12
    .asciz      "Hello"             @ 21

    .text
    .global     main
    .type       main, %function
main:
    stmfd      sp!, {fp,lr,v1,v2}
```

The pre-processing step also constructs a 'Symbol Table' (it is not technically a symbol table but is named as such because it provides crucial information required during the ARM generation).
The Symbol Table is populated with

1. all class info (the fields of each class and their offsets, the total size required if object needs to be created on heap with malloc).
2. all var info to easily check the type of each var
3. all method info so that information related to a method call is accessible to the caller
    a. Additional stack space required
    b. Mapping of variables to stack offset
    c. Number of v registers (callee registers) used so that it can be stored on stack to be popped back later upon exiting from the function

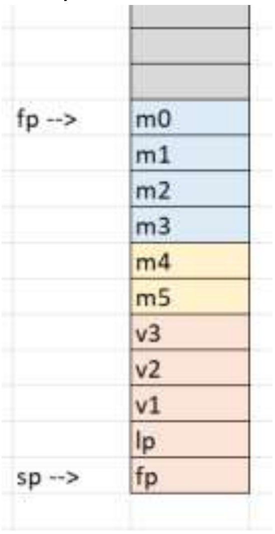
Relative Addressing on Stack

Stack space allocation:

- As there is a chance of a1 and a4 being modified in function calls. Their values are stored on the stack space m0 to m3 before branching to function calls so that they can be restored after returning from function call.
- Arguments that did not fit into a1-a4 are also stored on the stack
    - e.g. In add(this, 1,2,3,4,5), 4 and 5 are stored on m4 and m5 respectively
- Each memory address label mX refers to a specific offset from fp.
    - m0 is 0 bytes from fp
    - m1 is 4 bytes from fp
    - m2 is 8 bytes from fp

The image of the stack on the right following shows how the stack is managed

- The stack grows downwards (to smaller addresses)
- The frame pointer always points to the base of the stack frame and is most frequently used to retrieve data from the stack for the current activation frame
- The stack pointer is at the top of the stack and is used to prepare the stack for the next function invocation (store overflowing arguments on the stack for the next function call, then increment it before branching to the function



Translating IR3 to ARM Assembly Summary

IR3 Statement	ARM Assembly representation	ARM Example
Label	• The label itself	`.L3`

If	• Compare statement
	• Appropriate branch statement
		`cmp v1 #0`
		`bgt .L3`

Goto	• Branch to the label	`b .L3`

Println	• Load format specifiers and arguments onto arg registers
	• Branch to printf
		`mov a2, v1`
		`ldr a1, =LC0 + 3`
		`bl printf`

Readln	• Load format specifiers and write space onto arg registers
	• Branch to scanf
	• Load integer value into register
		`ldr a1, =LC0 + 0`
		`ldr a2, =LC0 + 12`
		`bl  scanf`
		`ldr a2, [a2]`

Return	• Load value onto a1 register
	• Branch to exit segment of function
		`mov a1, a4`
		`b   .Compute_2_exit`

AssignVarStmt	• Add ARM assembly code for RHS expression computation • Move value into LHS register	
AssignFieldStmt	• Prepare a temp unused register • Add ARM assembly code for RHS expression computation • Move value into LHS temp register • Store value in temp register to the respective offset from the object's address	
CallStmt / CallExp	• Store all a1-a4 registers that currently has a mapping to variable • Move the appropriate values into a1 to a4 arg registers • Store onto the stack arguments that cannot fit in a1 to a4 • Increment sp to the last value that was stored to the stack • Branch to function call • Decrement sp back by the same offset • Load original values of a1 to a4	

5 Challenges

The most challenging part is the back-end component as there are many edge cases to consider when allocating the registers. Due to the approach I took where registers are allocated method by method, block by block, and line by line, it can cause ambiguities when the register allocated to the same variable is different across lines. A possible instance is when there is a while loop that returns to an earlier block, and the starting block assigns a different register for a variable from the ending block of the loop body.

Another key issue for the ARM generator is the stack management. Careful consideration must be placed into deciding the amount of additional stack space needed by a method. Thereafter, the frame pointer and the stack pointer have to be adjusted correctly because otherwise, the stack data access could become buggy when the program returns from a function call.

Although extremely tedious, the front end and middle end components are relatively manageable.

6 Code Directory

```
├── README.md
├── doc
│   ├── LimYanPengGary_pa1_readme.md
│   └── LimYanPengGary_pa2_readme.md
├── run.sh       // bash script to run whole program
├── src
│   ├── compile.py      // Main entry point
│   ├── lex.py          // Lexer
│   ├── parse.py        // Parser - generates Parse Tree from tokens
│   ├── ast2.py         // AST generator and Type Checker
│   ├── ir3.py          // IR3 generator
│   ├── reg.py          // Register allocator
│   ├── arm.py          // ARM generator
│   └── constants.py    // Constants for ARM generator and reg allocator
└── test
    ├── parse
    │   ├── ... // List of assignment 1 test files
    ├── check
    │   ├── ... // List of assignment 2 test files
    ├── call.j
    ├── call.s
    ├── fib.j
    ├── fib.s
    ├── hello.j
    ├── hello.s
    ├── long.j
    ├── long.s
    ├── longer.j
    ├── longer.s
    ├── overload.j
    ├── overload.s
    ├── print.j
    ├── print.s
    ├── stmt.j
    ├── stmt.s
    ├── test_booleans.j
    ├── test_booleans.s
    ├── test_fields.j
    ├── test_fields.s
    ├── test_functions.j
    ├── test_functions.s
    ├── test_ops.j
    └── test_ops.s
```

To run the program, call
```
./src/compile.py filename.j
```
or

```
python ./src/gen.py filename.j
```
or
```
// Compiles and execute the ./test/fib.j file
run.sh fib
```