Intermediate-Code Generation

1 Overview

So far, we've completed the analysis part of the SPL compiler, and we are ready to move to the synthesis (or generation) part. Currently, your compiler can recognize a valid SPL program. In phase 3, your compiler will generate an *intermediate representation* (IR) for a given source program. The IR can be further optimized for better runtime performance.

Most compilers translate a source program first to some form of IR and then convert from there into machine code. The IR is a machine- and language- independent representation of the original source code. Although converting the code twice introduces additional costs, the use of an IR provides increased abstraction, cleaner separation between the front and the back ends. IR also facilitates advanced compiler optimizations. In fact, most optimizations are done on the intermediate code, rather than the source code or the target code.

In a broader sense, IR can be any form of the original program during compilation phases, such as the token stream generated by the lexical analyzer, the parse tree generated by the syntax analyzer, the three-address code produced in code generator, etc.When we talk about IR in this project, we mostly refer to three-address code (TAC), in which every single instruction can have at most **three** operands (addresses). We will elaborate on our TAC instruction set in Appendix E. We provide a simulator to execute the generated TACs, which is also installed in our lab virtual machine.

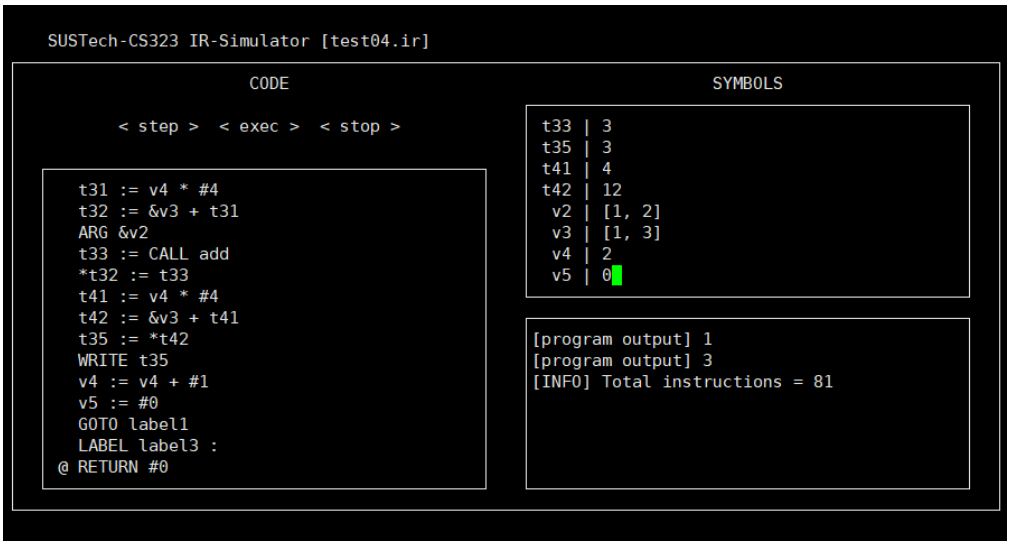
You will continue with code generation on top of your semantic analyzer. Please also be reminded to keep your code maintainable and extensible since the subsequent part of the SPL compiler relies on your work here.

2 Lab Environment

You will implement the code generator based on the last phase. Flex and Bison are still necessary. The dependencies are listed as follow:

- GCC version 7.4.0
- GNU Make version 4.1
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4
- Python version 3.6.8
- urwid (Python module) 2.0.1
- Spim version 8.0

We prepared an IR simulator so that you can run your generated TACs. It is available on our GitHub repository. It is built with a Python module, `urwid`. If you want to run it on your own computer, make sure you have installed `urwid`. Note that `urwid` **does not** support Windows environment. Besides, this simulator is buggy. Don't be surprised if you see abnormal behaviors :)



For example, if you want to run an IR program `test_a.ir` with three integers, 1, 9, and 42 as input, you may simply type the following command:

```
irsim test_a.ir -i 1,9,42
```

The option `-i` stands for input numbers, which are separated by commas (,). For a program without user input, this option can be simply ignored. You can specify a log file by the `-l` option. By default, log goes to the file `irsim.log` in the current working directory.

The simulator runs an IR program from the `main` function. There are three buttons in the panel, **step** will execute a single instruction, **exec** runs the program until the `main` function exits, and **stop** will terminate the current execution. The code section is under the three button. The instruction with a leading `@` sign is highlighted as the current program counter position. The right side of the simulator interface contains two panels. The one at the top records the runtime values of variables, while the one at the bottom displays program outputs. To exit the simulator, press the Esc key or SIGINT (^C) by Ctrl-C.

When the IR program terminates, the simulator will display the number of instructions it executed. You may use this number as an evaluation metric of the efficiency of your generated code. You should try your best to reduce the number of executed instructions.

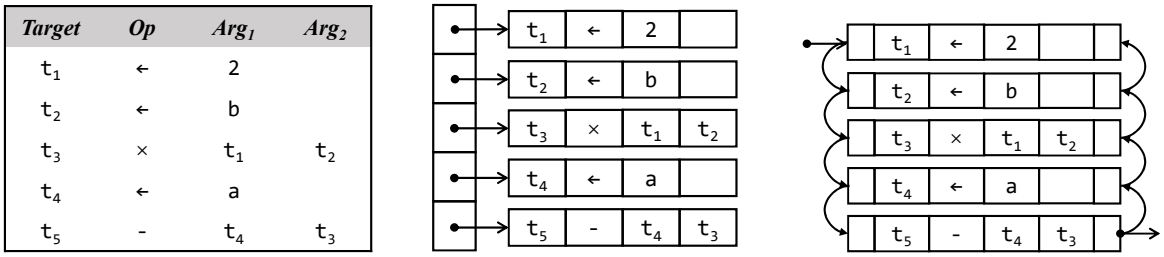
3 Intermediate Representation

In terms of representation format, there are linear IR, tree IR, graph IR, etc. In Phase 1, we have converted the source code into a parse tree, which is conceptually a form of tree IR.

The output in this phase, the TAC instructions, is a kind of *linear IR*. You can also organize the TAC instructions into a *graph IR*, in which a node represents a sequence of instructions without jumping, and an edge represents an unconditional/conditional jump between two basic blocks. This kind of IR is called the *control-flow graph*, which is helpful for many modern compiler optimizations, such as constant propagation and dead-code elimination.

Typically, you will not immediately print out the TAC segment once it is translated from some syntax nodes, since you may reorganize the code structure or do optimizations on the intermediate code. So you need to store them in the memory before printing them into the generated code file. For this purpose, it is important to consider how to store them with a proper data structure.

TAC is often stored in a set of *quadruples*. Each quadruple is an object/record that consists of four fields: an operator, two operands (sources), and a destination (result). To form blocks, the compiler needs a mechanism to connect individual quadruples. Figure 1 shows three different structures for implementing the TAC for the expression `a - 2 * b`.



(a) Static array style (b) Pointer array style (c) Doubly-linked list style

Figure 1: Three kinds of linear IR data structures

Figure 1a is the most straightforward implementation, in which a large array is allocated in advance. Each element in the array stores a single TAC instruction. However, it suffers from low efficiency. Every time a new instruction is inserted at the middle, you should pull all following elements one position behind. What's more, the code size is limited by the array's length. The pointer array implementation (Figure 1b) inherits the same limitation of the array implementation, excepts that moving elements is cheaper since only the addresses are swapped, rather than the list of quadruples. Doubly-linked list in Figure 1c is more recommended, though it is more complex than the previous two styles. Linked list data structure is efficient in both insertion, deletion and element replacement. Moreover, it does not require pre-allocated space, hence the code size can be arbitrarily large (as long as the memory is sufficient).

4 Runtime Environment

Programming languages provide abstractions to hardware details, such as names, scopes, data types, operators, functions, and control-flow constructs. However, modern computer

hardware can only understand low-level primitives, e.g., arithmetic operations on 32/64 bit integers, data movement and control structure by boolean expressions. Compilers must work with operating systems to support high-level abstractions on the target architectures.

To do so, a compiler cannot merely translate the code. It should also generate extra code to maintain high-level features by means of low-level operations. During code generation, the compiler should manage target-machine mechanisms so that the generated code can run on the same *runtime environment*. The runtime environment contains many details related to the target machine. In this phase, we only discuss about some general concepts and how they can be represented by our TAC specification. In the next phase, we will introduce more about the target runtime, i.e., MIPS architecture.

4.1 Data Representation

Many high-level languages provide various simple data types:

- **characters**: 1 or 2 (wide character) bytes
- **integers**: 2, 4 or 8 bytes
- **booleans**: 1 bit (typically use at least 1 full byte)

These types are directly supported by most hardware. In addition, the pointer types in C-family languages are stored as unsigned integers. Their ranges depend on the target architecture, i.e., 4 bytes for 32-bit machines that support maximum 4GB memory addressing, and 8 bytes for 64-bit machines. There is actually very little distinction between a pointer and an unsigned integer at low-level: they are all numbers, and the only difference is how the compiler interprets these numbers.

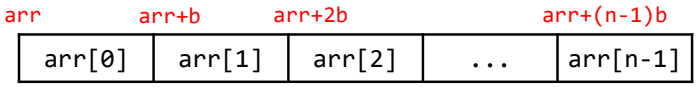
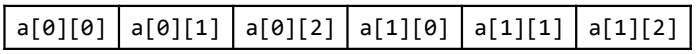


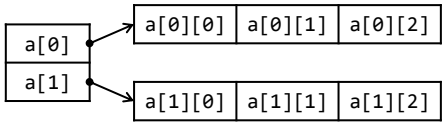
Figure 2: A C-style array in memory

An **array** stores a group of elements in continuous memory space. These elements are stored consecutively from low-address locations to high-address ones in the memory. Figure 2 shows a C-style array representation. The red labels represent the start address of each element, while `b` is the width of a single element. Recall that the name `arr` is a constant pointer, which points to the start address of the array. In general, for a 1-D array `arr` with `n` elements, the `i`-th (we count from 0) element is stored at the memory location `arr+i*b`. The runtime of SPL programs also adopts this kind of array representation.

There are two strategies to encode multi-dimensional arrays: true multi-dimensional array (Figure 3a) and array of arrays (Figure 3b). A true multi-dimensional array is one contiguous block. In a row-major configuration, the rows are stored one after another. For an $m \times n$ 32-bit integer array, the address of an element `arr[i][j]` is `arr + 4 * (n * i + j)`.



(a) True multi-dimensional array representation strategy



(b) Array of arrays representation strategy

Figure 3: Two strategies for representing multi-dimensional array

Alternatively, in the array of arrays representation, accessing an element involves two pointer dereferences, while the previous representation requires only once. Moreover, array of arrays is more space-consuming: for an $n_1 \times n_2 \cdots \times n_k$ array, the data fields is exactly the same as the true representation, while there will be extra $n_1 \times n_2 \cdots \times n_{k-1}$ pointer fields. The SPL runtime adopts the first strategy. Nonetheless, the second one is still considered advantageous for its flexibility.

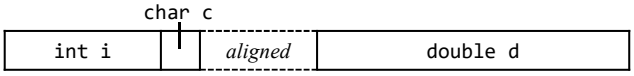


Figure 4: An aligned structure variable

A **Structure** variable is stored by allocating the fields sequentially in a contiguous block, then the member access is represented by the base address plus a member offset. The structure type `struct{int i; char c; double d;}` in C language requires 13 bytes with the individual fields at offsets 0, 4 and 5 bytes from the base address. However, it will actually allocate 16 bytes for this struct, because it will insert padding between the fields so that each field starts at an aligned boundary, as shown in Figure 4.

The SPL runtime is limited to integer type only, hence it is not essential to consider the data structure alignment. However, if you want to support more diverse data types, such as `char` or `short`, you should note that MIPS architecture requires all variables in the runtime memory aligned to the address length (4 bytes on a 32-bit machine).

4.2 Function Invocation

Each active function has its own *activation record*, which stores the information related to the function invocation. The information includes arguments, local variables, registers, return address, etc. In most architectures, the activation record is stored in the stack, hence a function's activation record is also called a *stack frame*.

In this project, you will not deal with the details about how to maintain the stack frame, since it is machine-dependent. You shall pass arguments explicitly with `ARG` statement, then invoke a function by its identifier with `CALL` statement.

There are two main approaches for passing arguments: *pass by value* and *pass by reference*. In the SPL runtime, primitive type arguments are passed by its true value. When a function is active, these values have their own copies on the stack frame, and changing the copies does not affect their original values, which are in the caller's frame. As for a derived type argument, the callee function accepts its reference, i.e., its starting address. In C language, we will explicitly pass a `struct` pointer to avoid copying the whole structure, while in SPL, `struct` arguments are implicitly converted to their starting addresses. This runtime behavior should be maintained by your compiler: to pass a `struct` variable `s1`, you should push it by `ARG &s1` rather than `ARG s1`; to access a member at a particular offset, you should add the base address by the offset and then access the new address with the dereference operator `*`.

What's the reason of pushing arguments in reverse order (see Appendix)? The answer is related to the memory layout of the calling sequence. Take Linux-x86 runtime as an example:

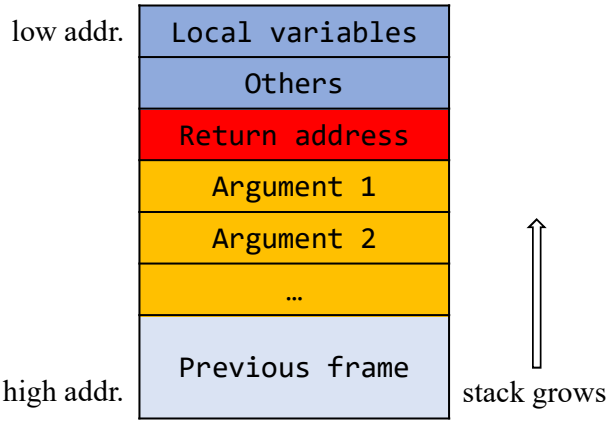


Figure 5: A runtime stack of a Linux-x86 process

When a function is invoked, a new frame will be pushed into the stack. Here, the "push" action contains multiple steps: firstly, arguments are pushed (from high address to lower), then the return address is recorded on top of the arguments, and finally, the memory units for local variables are prepared. The memory section from the local variables to the arguments make up the new stack frame for the callee.

Since arguments are pushed one by one, the first pushed argument is at the bottom of the stack (high address), and the last one is at the top (low address). That's why we should push them in reverse order. In this way, fetching arguments in the function body is straightforward: arguments will be represented by the offset from the return address register. In the example, argument 1 can be represented by `%ret+4`, and argument `i` at `%ret+4*i`[1]. Here, 4 (bytes) is the address length on a 32-bit machine.

[1] Actually, there is no `%ret` register, but the `%ebp` register stores `%ret` by a constant offset

5 Translation Schemes

To generate TAC from a parse tree, you need to traverse the tree in **post-order**, then convert the tree nodes according to some particular patterns. You will resort to *syntax-directed translation* to accomplish this task. At the implementation level, you have to implement a set of `translate_X` functions for each nonterminal `X`, and invoke these functions recursively, either directly or indirectly. We will introduce some examples that translate particular productions, though there may be other ways of translation, such as constructing the corresponding control flow graph and linking basic blocks by jump instructions.

5.1 Expressions

We categorize expressions into basic and conditional expressions. They occupy a portion of the 26 rewriting rules of the grammar symbol `Exp` in our SPL language specification. This section lists some of their translation schemes.

5.1.1 Basic Expressions

Table 1 shows translation schemes for basic expressions. We assume that the symbol table, which you have built in Phase 2, is globally accessible. Function `symtab_lookup` accepts a string of a variable's identifier, and returns the corresponding symbol information. The parameter `place` is the address variable that stores the evaluation result of this expression. `translation_Exp` returns TAC code of the node `Exp` and its children nodes.

In the right column, the code enclosed by a pair of brackets represents a concrete TAC instruction, and using plus sign (`` `+' ``) on two TAC instructions means concatenating them.

The translation schemes for the first two rewriting rules are simple. For the production of assignment expressions, We've already constrained the lvalue to be an identifier, array expression, or structure expression through semantic analysis. Here we only show the case when the left side is an identifier, i.e., $\texttt{Exp}_1$ `-> ID`, and we will talk more about the last two cases later in § 5.4. Recall that assignment operator is right associative, so we should firstly evaluate $\texttt{Exp}_2$, then assign the resulting value to the corresponding variable.

For arithmetic operations, we should evaluate two operands, then apply the corresponding operator on them. Here, we define a left-to-right evaluation ordering, hence we evaluate $\texttt{Exp}_1$ before $\texttt{Exp}_2$. The unary minus of an expression results in a negative case, which can be simply represented by subtracting itself from zero.

There are 9 (3 booleans and 6 comparisons) conditional expressions defined in SPL grammar. Their values are either 0 or 1. We translate these expressions by an auxiliary function `translate_cond_Exp`, which will be illustrated in § 5.1.2.

5.1.2 Conditional Expressions

The conditional expressions are typically used in `if-` and `while-` statements, which involve conditional jump instructions. These jump instructions will be generated together with the

Table 1: Basic expression translation schemes

`translate_Exp(Exp, place)` = case `Exp` of	
`INT`	`value = to_int(INT)` `return [place := #value]`
`ID`	`variable = symtab_lookup(ID)` `return [place := variable.name]`

$\texttt{Exp}_1$ `ASSIGN` $\texttt{Exp}_2$	`variable = symtab_lookup(`$\texttt{Exp}_1$`.ID)`
	`tp = new_place()`
	`code1 = translate_Exp(`$\texttt{Exp}_2$`, tp)`
	`code2 = [variable.name := tp]`
	`code3 = [place := variable.name]`
	`return code1 + code2 + code3`

$\texttt{Exp}_1$ `PLUS` $\texttt{Exp}_2$	`t1 = new_place()`
	`t2 = new_place()`
	`code1 = translate_Exp(`$\texttt{Exp}_1$`, t1)`
	`code2 = translate_Exp(`$\texttt{Exp}_2$`, t2)`
	`code3 = [place := t1 + t2]`
	`return code1 + code2 + code3`

`MINUS Exp`	`tp = new_place()`
	`code1 = translate_Exp(Exp, tp)`
	`code2 = [place := #0 - tp]`
	`return code1 + code2`

cond. `Exp`	`lb1 = new_label()`
	`lb2 = new_label()`
	`code0 = [place := #0]`
	`code1 = translate_cond_Exp(Exp, lb1, lb2)`
	`code2 = [LABEL lb1] + [place := #1]`
	`return code0 + code1 + code2 + [LABEL lb2]`

conditional expression's IR code. We define an auxiliary function `translate_cond_Exp`, which takes two additional parameters, `lb_t` specifies the location (label) to jump when the condition evaluates to *TRUE* (1 in its essence), and `lb_f` for the *FALSE* case, to translate a conditional expression. You should ensure the first argument of `translate_cond_Exp` is always a conditional expression. This can be checked before translation.

The schemes in Table 2 produce *short-circuit evaluations* for operation `AND/OR`. To see the effect, consider rule $\texttt{Exp}_1$ `AND` $\texttt{Exp}_2$: when $\texttt{Exp}_1$ evaluates to *FALSE* (i.e., 0) by the IR code of `translate_cond_Exp(`$\texttt{Exp}_1$`, lb1, lb_f)`, the control flow jumps directly to `lb_f` and `code2` is skipped. For the case it is *TRUE*, `code2` will be executed to evaluate the whole `AND`-expression. The same mechanism holds for `OR`-expressions.

We list translation schemes for 4 conditional operations (for the other cases, simply replace

Table 2: Conditional expression translation schemes

`translate_cond_Exp(Exp, lb_t, lb_f)` = case `Exp` of	

$\texttt{Exp}_1$ `EQ` $\texttt{Exp}_2$	`t1 = new_place()`
	`t2 = new_place()`
	`code1 = translate_Exp(`$\texttt{Exp}_1$`, t1)`
	`code2 = translate_Exp(`$\texttt{Exp}_2$`, t2)`
	`code3 = [IF t1 == t2 GOTO lb_t] + [GOTO lb_f]`
	`return code1 + code2 + code3`

$\texttt{Exp}_1$ `AND` $\texttt{Exp}_2$	`lb1 = new_label()`
	`code1 = translate_cond_Exp(`$\texttt{Exp}_1$`, lb1, lb_f) + [LABEL lb1]`
	`code2 = translate_cond_Exp(`$\texttt{Exp}_2$`, lb_t, lb_f)`
	`return code1 + code2`

$\texttt{Exp}_1$ `OR` $\texttt{Exp}_2$	`lb1 = new_label()`
	`code1 = translate_cond_Exp(`$\texttt{Exp}_1$`, lb_t, lb1) + [LABEL lb1]`
	`code2 = translate_cond_Exp(`$\texttt{Exp}_2$`, lb_t, lb_f)`
	`return code1 + code2`

`NOT Exp`	`return translate_cond_Exp(Exp, lb_f, lb_t)`

`EQ` token). In fact, since C is weakly typed, non-conditional expressions can also be used in control flow statements. A widely used example is `while(T--){}` block, where the loop body will execute `T` times for any non-negative integer `T`, and the loop terminates when `T` reaches zero. You can think about how to translate this kind of control flow, and realize it as your compiler's extra feature.

5.2 Statements

Table 3 shows translation schemes for statements. To translate control-flow statements (i.e., `if-` and `while-`), we utilize the previously defined function `translate_cond_Exp`. Translating conditional expressions introduces conditional jumps, while there are only unconditional jumps in statement's translation schemes.

5.3 Function Invocations

There is one thing to do before translating the functions: you should add a `read` function to the symbol table, which takes no parameter and returns an integer value, and a `write` function that accepts a single integer parameter. These two pre-defined functions provide user-interaction (I/O) for SPL programs. They have special translation schemes: `read` function invocation should be translated into the `READ` instruction, and `write` to the `WRITE` instruction, similarly.

There are two rewriting rules related to function invocation, the one with arguments and the other without. The built-in function `read` and `write` have their own translation schemes.

Table 3: Statement translation schemes

translate_Stmt(Stmt) = case Stmt of	
RETURN Exp SEMI	tp = new_place() code = translate_Exp(Exp, tp) return code + [RETURN tp]
IF LP Exp RP Stmt	lb1 = new_label() lb2 = new_label() code1 = translate_cond_Exp(Exp, lb1, lb2) + [LABEL lb1] code2 = translate_Stmt(Stmt) + [LABEL lb2] return code1 + code2
IF LP Exp RP Stmt_1 ELSE Stmt_2	lb1 = new_label() lb2 = new_label() lb3 = new_label() code1 = translate_cond_Exp(Exp, lb1, lb2) + [LABEL lb1] code2 = translate_Stmt(Stmt_1) + [GOTO lb3] + [LABEL lb2] code3 = translate_Stmt(Stmt_2) + [LABEL lb3] return code1 + code2 + code3
WHILE LP Exp RP Stmt	lb1 = new_label() lb2 = new_label() lb3 = new_label() code1 = [LABEL lb1] + translate_cond_Exp(Exp, lb2, lb3) code2 = [LABEL lb2] + translate_Stmt(Stmt) + [GOTO lb1] return code1 + code2 + [LABEL lb3]

They are shown at the first two rows of Table 4. Note that `write` function takes only one argument, hence we rewrite the `Args` symbol directly to `Exp`.

Let's revisit the arguments passing order in the SPL runtime. We've already mentioned (§ 4.2) that, the arguments should be pushed into the stack in reverse order as they are declared. In the translation scheme of `Args`, we accomplish this behavior by passing an additional `arg_list` parameter. Each time a new argument is evaluated, its reference is inserted at the head of this list. Before inserting the `CALL` instruction, we pop these references from `arg_list` and generate their corresponding `ARG` instructions. Here, we simulate a first-in-last-out list, hence the arguments for a callee function are evaluated from left to right, and they will be pushed into the runtime stack from right-to-left.

It's worth mentioning that the order of evaluation[2] of C language is an **undefined behavior**. The function arguments are specified by comma-expression, and a compiler can evaluate each sub-expression in an arbitrary order[3]. For example, to invoke `foo(a+b, a-b,`

[2] https://en.cppreference.com/w/c/language/eval_order

[3] The order of evaluation is different to the concept of associativity: the comma operator is left-to-right associated, but its evaluation order is undefined

Table 4: Function invocation translation schemes

`translate_Exp(Exp, place)` = case `Exp` of	
read `LP RP`	`return [READ place]`
write `LP Exp RP`	`tp = new_place()` `return translate_Exp(Exp, tp) + [WRITE tp]`
`ID LP RP`	`function = symtab_lookup(ID)` `return [place := CALL function.name]`

`ID LP Args RP`	`function = symtab_lookup(ID)`
	`arg_list = EMPTY_LIST`
	`code1 = translate_Args(Args, arg_list)`
	`code2 = EMPTY_CODE`
	`for i = 1 to arg_list.length:`
	`  code2 = code2 + [ARG arg_list[i]]`
	`return code1 + code2 + [place := CALL function.name]`

`translate_Args(Args, arg_list)` = case `Args` of	

`Exp`	`tp = new_place()`
	`code = translate_Exp(Exp, tp)`
	`arg_list = tp + arg_list`
	`return code`

`Exp COMMA Args`	`tp = new_place()`
	`code1 = translate_Exp(Exp, tp)`
	`arg_list = tp + arg_list`
	`code2 = translate_Args(Args, arg_list)`
	`return code1 + code2`

`-a)`, a compiler can generate code that evaluates from left-to-right, from right-to-left, or even evaluates `a-b` first, then `-a`, and then `a+b`. In the provided translation scheme, we define a left-to-right evaluation order for function arguments. However, it is up to you to generate code with another evaluation order. The only important matter is that the arguments should be passed in a proper order.

5.4 Derived Data Types

There are two derived date types in SPL: arrays and structures. Unlike integers, of which the width is fixed, we should dynamically allocate memory before using variables of derived types, by using `DEC` statement with the size of the declared type. Generally, derived data types can be treated as collections of elements. Accessing a particular element can be represented by the form `base + offset`, where `offset` is calculated from the size of preceding elements. To use a derived type variable, there are two rewriting rules: `Exp -> Exp LB Exp RB` and `Exp -> Exp DOT ID`, both of which will be translated into memory offset calculation.

We adopt true multi-dimensional array representation, which stores all elements in a

single consecutive block. In an SPL array, all elements are equal in size, and the sizes of all arrays in the same lower dimension are also equal. So we can calculate the address of an element by (assuming that the data type of the array element is T):

$$ADDR(arr[i_1]\dots[i_n]) = ADDR(arr) + OFFSET(arr[i_1]\dots[i_n]) \times SIZE(T)$$

The *OFFSET* function does not involve the data size. It only depends on the number of preceding elements. Take a 3-dimensional array `arr[l][m][n]` as an example, the offset of element `arr[i][j][k]` is:

$$OFFSET(arr[i][i][k]) = i \times m \times n + j \times n + k$$

The above formula can be easily generalized to arrays with higher dimensions.

For structure variable, the offset is calculated by the sum of preceding elements' sizes. Though we assume all variables are integers, you can design your own way to represent other data types, such as character, or short or even floating-point numbers[4]. If the fields in a structure variable are of different types, we cannot calculate the offset and multiply it with a constant size. The formula of addressing a particular field is:

$$ADDR(st.f_i) = ADDR(st) + \sum_{t}^{i-1} SIZE(st.f_t)$$

However, you should consider the alignment. In the runtime, all structure fields should be aligned to 32-bit boundary. To simplify the implementation, you can allocate all primitive types in 4 bytes.

There are more complex cases, such as accessing an array element, while the array is a field of a structure. With a proper data type representation and program design pattern, dealing with such complex cases is not very difficult. Since you are not required to implement derived type variables, we will not give you the concrete translation schemes. Nevertheless, you can implement this feature to obtain a higher score.

5.5 Hints

We recommend you to define a separate class for intermediate representation (collection of TAC instructions, see §3 for possible implementation choices). Then, you may construct a sample TAC program artificially and run it on the IR simulator to verify your IR class.

You need to implement the `translate_X` function for each grammar symbol `X`. We have introduced translation schemes for expressions and statements. You should think about how to deal with other cases such as data initialization, structure member accessing, etc. Once you have understood and realized the translation schemes introduced previously, it should be simple to write new schemes. Writing more test cases by your own always helps find your compiler's inadequacy.

[4]You need to convert them into integers, since the IR simulator only supports integers

6 Requirements

6.1 Basic Requirements

In this phase, the input format is the same as the previous phases, i.e., the executable `splc` accepts a single command line argument representing the SPL program path. Our test cases are free from lexical/syntax/semantic errors, so your task is to generate TAC-IR that preserves the behavior of this program without considering reporting errors (though you are free to keep those checking routines). You should carefully read the assumptions in §6.2 to avoid getting stuck at over-complicated situations (such as structures with arrays as fields or arrays of structures, etc.).

You should compile your IR generator as the `splc` target in `Makefile`, and move the executable to the `bin` directory. For example, the `Makefile` is placed under the project root directory, and we make the `splc` target by:

```
make splc
```

which generates the executable. Then we run it by:

```
bin/splc test/test_3_r01.spl
```

The corresponding TAC instructions should be printed in text file `test/test_3_r01.ir`, and you may verify the generated IR using the provided IR simulator.

6.2 Assumptions

Below are the assumptions for our test cases, which means that you can implement your code generator safely by ignoring their violations. They are:

Assumption 1 all tests are free of lexical/syntax/semantic errors (suppose there are also no logical errors)

Assumption 2 there are only integer primitive type variables

Assumption 3 there are no global variables, and all identifiers are unique

Assumption 4 the only return data type for all functions is `int`

Assumption 5 all functions are defined directly without declaration

Assumption 6 there are no structure variables or arrays

Assumption 7 function's parameters are never structures or arrays

6.3 Input and Output

Your code generator shall translate the input SPL program into the TAC instructions. To show how it works, see the following SPL program:

Listing 1: SPL test case for intermediate-code generation

```
1  int main()
2  {
3      int n;
4      n = read();
5      if (n > 0) write(1);
6      else if (n < 0) write(-1);
7      else write(0);
8      return 0;
9  }
```

The code is a signum function, which outputs 1 for positive numbers, -1 for negative numbers, and 0 for zero. A valid intermediate representation of this program is:

```
FUNCTION main :
READ t1
v1 := t1
t2 := #0
IF v1 > t2 GOTO label1
GOTO label2
LABEL label1 :
t3 := #1
WRITE t3
GOTO label3
LABEL label2 :
t4 := #0
IF v1 < t4 GOTO label4
GOTO label5
LABEL label4 :
t5 := #1
t6 := #0 - t5
WRITE t6
GOTO label6
LABEL label5 :
t7 := #0
WRITE t7
LABEL label6 :
LABEL label3 :
t8 := #0
RETURN t8
```

Our sample output adopts the naming convention that variable names follow the pattern `t`*`n`* or `v`*`n`*, and `label`*`n`* for label names. However, this is not the only way. Your compiler can generate any valid names as you wish.

Note that the above TAC IR is far from efficient. For example, it assigns constant 0 to four variables `t2`, `t4`, `t7`, and `t8`, while they can be the same local variable. Also, there are dummy labels like `label6`. You may build a compiler that generates more efficient code.

6.4 Optional Features

You are encouraged to design and implement your own language features. However, to ensure that the generated code can run safely in the IR simulator, you should carefully design the translation schemes so that the extra features can be supported by the given TAC instructions. A suggested optional feature is to support continue/break statements in loops. You can easily extend SPL grammar by adding new terminals, and translate the constructs into some kinds of `GOTO` structures.

You can also implement your compiler by modifying the assumptions, for example:

- modify **Assumption 6** and **Assumption 7**, so that structure variables can appear in the program, and they can be declared as function parameters. Still, assignment operations will not be directly performed on a structure variable.
- modify **Assumption 6** and **Assumption 7**, so that 1-D arrays can be declared as function parameters, and multi-dimensional arrays can be defined as local variables

Appendix

E Three-Address Code Specification

Table 5 defines the instruction set of the three-address code used in our SPL compiler. Your generated TAC should strictly follow the specification, otherwise, our IR simulator cannot recognize your generated code. Two important things to note: (1) the empty space cannot be eliminated, particularly, there must be an empty space between the label/function name and the colon (the first two rows); (2) all keywords (`LABEL`, `CALL`, etc.) must be in upper case, or they will be recognized as identifiers.

Table 5: Three-address-code specification

Instruction	Description
`LABEL x :`	define a label `x`
`FUNCTION f :`	define a function `f`
`x := y`	assign value of `y` to `x`
`x := y + z`	arithmetic addition
`x := y - z`	arithmetic subtraction
`x := y * z`	arithmetic multiplication
`x := y / z`	arithmetic division
`x := &y`	assign address of `y` to `x`
`x := *y`	assign value stored in address `y` to `x`
`*x := y`	copy value `y` to address `x`
`GOTO x`	unconditional jump to label `x`
`IF x [relop] y GOTO z`	if the condition (binary boolean) is true, jump to label `z`
`RETURN x`	exit the current function and return value `x`
`DEC x [size]`	allocate space pointed by `x`, `size` must be a multiple of 4
`PARAM x`	declare a function parameter
`ARG x`	pass argument `x`
`x := CALL f`	call a function, assign the return value to `x`
`READ x`	read `x` from console
`WRITE x`	print the value of `x` to console

We explain the instructions in detail below:

- Keyword `LABEL` defines a target, which can be jumped to by the `GOTO` statement
- `FUNCTION` specifies a set of instructions as a function. The function definition ends when a new function is defined, or upon reaching end-of-file
- Assignment operation assigns the value on its right side (rvalue) to the left side symbol (lvalue). An lvalue can only be a variable, while the rvalue can be either variable or immediate value. An immediate value is specified by the form `#n`. For example, the TAC instruction `t1 := #9` means assign constant value 9 to the variable `t1`

- There are four arithmetic operations in the TAC, addition, subtraction, multiplication, division, the operands can be either variables or immediate values
- The `&` symbol stands for "address of", which returns the address of a variable. For example, the instruction `b := &a + #8` means adding the address of `a` by 8, then assign the new address to `b`
- If an dereference operator (`*`) is attached to a right-side variable in an assignment operation, it returns the value stored at that address. When `*x` appears in the left side of an assignment statement, it means that the rvalue should be copied to the address `x`
- There are two types of `GOTO` (jump) statements, unconditional jump and conditional jump. An unconditional jump to `x` will move the program counter to the line right after `LABEL x`. As for conditional jump, if the boolean expression evaluates to *true*, then the `GOTO` part executes, otherwise, the program counter moves to the next instruction. A relational operator `[relop]` can be any of the six: `<`, `<=`, `>`, `>=`, `!=`, `==`
- `RETURN` statement moves the current program counter from a function to its caller, and returns a value `x`, which can be either a variable or an immediate value
- The keyword `DEC` is designed for allocating consecutive memory. It is useful for derived type variables such as array or structure. The unit of `[size]` is byte, so an integer array of size 10 should take 40 bytes in memory. For those primitive type (`int`) variables, there is no need to allocate memory using `DEC`
- The following three instructions are related to function invocation. `PARAM` is used for declaring function parameters, it follows the `FUNCTION` statement. For example, if function `foo` has three parameters, its declaration is translated into:

  ```
  FUNCTION foo :
  PARAM p1
  PARAM p2
  PARAM p3
  ```

  To call this function, you should pass the arguments one-by-one in reverse order, then apply the `CALL` statement:

  ```
  ARG a3
  ARG a2
  ARG a1
  t := CALL foo
  ```

- `READ` and `WRITE` are designed for user interaction. In our IR simulator, `READ` statement can read a user-input integer from the console, and `WRITE` prints an integer value to the console