CS 4240: Compilers and Interpreters
Project Phase 1: Front end

Total points: 150

Due Date: October 22nd 2018 (11:59 pm EST) (via Canvas)

Introduction

This semester, through a project split into 2 phases, we are going to build a full compiler for a small language called Tiger targeted towards the MIPS machine. As discussed, the phases of the project will be:

- Phase I (Front end): parser for Tiger language, symbol table and semantic analysis and intermediate representation (IR) code generation
- Phase II (Back end): Instruction selection, register allocation and MIPS code generation

This phase (phase 1) is further split into two parts: part I deals with building a scanner and a parser and detecting syntax errors. Part II deals with the semantics analysis and IR code generation.

Part I Description

The purpose of this part is to build a parser for the Tiger language that uses a longest match scanner and LL parser; both will be generated using the ANTLR v4 (https://github.com/antlr/antlr4/blob/master/doc/getting-started.md) (exact version to be used will be notified by the TA, a Vagrant build of the same may be given to you). First you will study the ANTLR using lots of examples and documentation that are provided with it. ANTLR v4 is a production tool used by professional and it can generate lexer and parser in C++ or Java as output using the input lexical and grammatical specifications. After studying a few examples and reading the documentation, we are now ready to tackle building the compiler front end using ANTLR.

Please refer to the language specifications for Tiger given in Appendix A. It is a small language with properties that you are familiar with: functions, arrays, records, integer and float types, control flow, etc. – the syntax and semantics of the Tiger's language constructs are described in details in the appendix A of the document along with a sample program.

First, we need to build a scanner that will scan the input file containing the Tiger program and perform lexical actions and return tokens one by one on demand to the parser. You will first test it by writing Tiger programs as per the grammar, invoke the scanner through the parser and check the stream of the tokens generated. You can build the lexer using ANTLR

1.A : Lexical Definition

The following token types are possible in Tiger:

MAIN COMMA COLON SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE PERIOD PLUS MINUS MULT DIV EXP EQ NEQ LESSER GREATER LESSEREQ GREATEREQ AND OR ASSIGN

Keyword tokens: MAIN ARRAY RECORD BREAK DO ELSE END FOR FUNC IF IN LET OF THEN TO TYPE VAR WHILE ENDIF BEGIN END ENDDO

ID INTLIT FLOATLIT

It may be noted that the first block above includes all punctuations; the second block above includes all keywords and the third block includes user-defined identifiers, integer and float literals.

Notes:

- The keywords are recognized as a subset of the identifiers – that is first the scanner recognizes an identifier (ID) and then check the string against a list of keywords, if it matches you return corresponding keyword token and not an ID.
- The scanner uses the longest match algorithm when recognizing tokens. That is, the scanner keeps matching the input character to the current token, until you encounter one which is not a part of the current token. At this point, the token is completed using the last legal character and is returned to the parser. Next time around, the token generation restarts from the first character which was not a part of the current token. Lesson 1 slides demonstrate the working of this algorithm on "main".

1.B : Scanner details

The scanner reads in char by char from the input file and perform token generation per longest match algorithm. Thus, it is able to read in a stream of characters and, return the next matched <token type, token> tuple, or it throws an error. For example, given the stream "var x := 1 + 1", the first request to the scanner for a matching token returns <VAR, "var">, the second call <ID, "x">, third call <ASSIGN, ":=">, next <INTLIT, "1">, <PLUS, "+">, and so forth.

The scanner implements lexical specification of Tiger: It reads in the program's stream of characters and return the correct token tuple on each request. For lexically malformed Tiger programs, the scanner throws an error which prints: line number in the file – the partial prefix of the erronous string (from the input file), the malformed token putting it in quotes pin-pointing the culprit character that caused the error. The scanner is capable of catching multiple errors in one pass – ie it does not quit but continue on after catching the 1st error. It will throw away the bad characters and restart the token generation from the next one that starts a legal token in Tiger.

Implementation: You will write the lexical specification of Tiger in a form acceptable to ANTLR v4 and generate the lexer program in C++ or Java as per your choice of implementation. You will then test it for both errors and correct production of stream of tokens. As far as the errors are concerned, you can just produce the errors generated by the ANTLR generated lexer.

1.C : Parser

You will be first writing a parser for Tiger that calls the scanner above which supplies it the tokens. This parser too will be automatically built by using ANTLR. The output parser should be LL(k), by rewriting the grammar k should be minimized, possibly to 1. For this purpose you will be rewriting the grammar that goes as an input to the ANTLR. This consists of three parts:

1. Rewrite the grammar given in the Tiger language specification below to remove the ambiguity by enforcing operator precedences and left and right associativity for different operators. This part is to be done by hand.
2. Modifying the grammar obtained in step 1 to support LL(k) parsing, k must be minimized, so that the parser is LL(1). This could include removing left recursion and performing left factoring on the grammar obtained in step 1 above. You are not allowed to rewrite the grammar except using these two techniques. This part is to be done by hand.
3. Once you have the grammar in the correct form, you will input it to ANTLR and it generate the parser in C++ or Java as per your choice. You will then test the parser feeding it Tiger programs which are used as test cases. You can iterate and revise the grammar repeating steps 1 and 2 until you get it right. The generated parser when invoked on input Tiger program will generate a parse tree which can be potentially visualized with a suitable IDE plug-in – refer to : http://www.antlr.org/tools.html

For syntactically correct Tiger programs, the parser should output "successful parse" to stdout. For programs with lexical issues, the scanner is already responsible for throwing an error. For programs with syntactic problems, however, the parser is responsible for raising its own errors. In these cases, the output should be some reasonable message about the error, which should include : input file line number where it occurred, a partial sentence which is a prefix of the error, the erroneous token and perhaps what the parser was expecting there. In addition, your parser should also output the sequence of token types it sees, as it receives them from the scanner when you turn on a debug flag in your code. This will help us in verifying your solution. For example, given the stream "var x := 1 + 1", the parser would output "VAR ID ASSIGN INTLIT PLUS INTLIT". You will just use the error recovery mechanisms provided in the ANTLR and test and report their outcomes on different input test cases. More advanced error recovery mechanisms than the above are not expected to be built and are out of scope of this project.

1.D : Turn-in

Correctness

You will be provided with some simple programs for testing. You will also be provided with several test inputs.

Grading

Deliverables for part 1:

1. Hand-modified Tiger grammar in appropriate LL(k) grammar form (25 points)

2. Generated Parser code (15 points)
3. Testing and output report (10 points)

Part II : Symbol Table, Semantic Analysis, IR Code generation

This part of the project consists of three sub-parts. You will be creating a symbol table, doing semantic checks, and generating intermediate code for use in the final phase. For all these purposes you will be using the ParseTree Walking mechanisms provided by the ANTLR (use of listener and visitor objects -https://github.com/antlr/antlr4/blob/4.7.1/doc/listeners.md
First study these mechanisms by going over the relevant parts in ANTLR book and by studying the examples – you can then add suitable code to respective objects to first implement symbol table generation and then simple parts of speech (such as expression trees consisting of simple operators followed by complex ones, advancing toward more and more complex sentences and parts of speech).

II.A: Symbol Table

The symbol table is a useful structure generated by the compiler. Semantic checking (part 2) will make extensive use of it. The symbol table holds declarative information about variables, constants, user defined types, functions and their expected parameters and so on that make up a program. The following are the typical entries (this is not an exhaustive list) for a symbol table:

- variable names
- defined constants
- defined types
- procedure and function names and their parameters
- literal constants and strings
- source text labels
- compiler-generated temporaries

You must also the attributes for each of the above entries. Typical attributes include:

- textual name
- data type
- dimension information (for aggregates), array bounds
- declaring procedure
- lexical level of declaration (scoping)
- storage class (base address)
- offset in storage
- if record, number of fields and a list of fields with underlying types
- if parameter, by-reference or by-value?

- can it be aliased? to what other names?
- number and type of arguments to functions

You have some flexibility in deciding these implementation details, and the above lists are by no means requirements. The guiding principle you should use for building your symbol table and the information you keep in it should be that of utility: Store what you will need for the next parts – esp. the semantic checking and IR code generation. Your implementation will likely make use of one or more hash-tables to implement a symbol table.

Scoping is a key aspect of symbol tables. By this we mean the following two things:

1. The most closely nested rule (i.e. references always apply to the most closely nested block) – please implement the scoping rules described in the language definition.
2. Declaration before use

Thus, insertion into your symbol table cannot overwrite previous declarations but instead must mask them. Subsequent lookup operations that you run against the symbol (as you do type checking in part 3, for example) should return the most recently inserted variable definition, etc. This handles rule 1. For rule 2, a lookup operation should never fail. If it does, it means the symbol table has not yet seen a declaration for the reference you are attempting to check and it is a semantic error to use an undeclared value.
The scoping rule we are going to use is the same as used in all block structured languages: inner declaration of a variable name hides the outer declaration; in a given scope, the innermost declaration is the one that is visible and a name used in that scope is bound to that declaration.

As you consider scoping, you are free to implement your symbol table on a scope by scope basis or a a global symbol table in which declarations are entered upon entering a scope and deleted upon exiting it. You can use chaining for the entities mapped to the same hash location. Follow scoping rules of Tiger. You should read the relevant symbol table design material from the Cooper's book.
Here are the suggested steps:

1. First carefully read the grammatical and semantic specification of Tiger and decide which values and which of their attributes are going to be held in symbol table.
2. Design a symbol table with hash maps, chains, etc.
3. Implement the symbol table generation

For the above purposes you will implement the listener and visitor objects that interface to the parse tree (refer to : https://github.com/antlr/antlr4/blob/master/doc/listeners.md and write suitable tree walking code to gather and then store the relevant information into the symbol table as discussed above.

II.B: Semantic Checking

This phase consists of semantic checks. It implements semantic checks by walking the Parse Tree by using listener and visitor objects. You will develop the necessary walking and checking code here. You should first read the semantic specification of Tiger and then implement the checking as follows.

Typically, the first step in semantic checking is the binding analysis – to determine which variable name binds to which symbol table entry. This is done using look-up mechanism as per the block structure rule. The first step is to decorate the respective nodes of the Parse tree with attributes elicited from the symbol table. The usage of a variable must occur as per its declaration, e.g., number of fields and their usage names must match their declarations for a record etc. Finally, the type checking is done. Some checks might involve making sure the type and number of parameters of a function match actual arguments. There are several cases in Tiger where type checking must occur:

- Agreement between binary operands
- Agreement between function return values and the function's return type
- Agreement between function calls and the function's parameters

Refer to the Tiger reference manual for many rules about the type semantics: the "Operators" section discusses types with respect to binary operators; the "Control Flow" section states that the if ,while, and for expression headers should all evaluate to true or false; the "Types" section dictates that you enforce a "name type equivalence" for your types; etc. The other semantic check that must be made is with respect to return statements: A function with no return type cannot have a return statement in it.

For correct programs, your compiler should pass this part silently and emit nothing. For programs with semantic problems, your compiler should emit an error message stating the problem and relative place in the source.

II.C: Intermediate Code

The final part of this phase is to convert the program into intermediate code. For the purposes of this project, we will be using 4-address code (or "quadruple" 3-address), which has the following form:

```
op, y, z, x
```

This reads as, "Do operation op to the values y and z, and assign the new value to x." A

simple example of this can be given with the
following: 2 * a + (b - 3)

The IR code for this expression would be
following:

```
sub, b, 3, t1
mult a, 2,t2
add t1,t2,t3
```

As we can see, an important characteristic of this representation is the introduction of temporary variables, which the compiler (and therefore you, as the compiler writer) must generate. Notice that this also implies that temporary variables must now be made part of the symbol table, and their type should be derived from the result of the operation. For example, in the above expression, t1, t2, and t3 would all marked as type int. We say that the types for temps "propagate" through the program. You do not yet have to worry about the number of temporary variables you are creating. That is, you may assume infinitely many temporary names are at your disposal. First read the lesson on ircodegen (IR code generation) and then implement the necessary tree walks and generations of labels and temporaries and generate the quad IR. In the IR code, you will annotate the types of the operands (such as t1.int etc.) and also operators (such as add.int) wherever necessary. You will need this information for allocating storage as well as for selecting the right instructions (code generation) in phase 2.

One subtlety with the intermediate code is handling arrays. Arrays may be multidimensional, so you will have to linearize accesses. Consider the following examples for how we expect you to calculate the offsets:

- A one dimensional array of size 5, accessed by "arr[1]" → offset is 1
- A two dimensional array of size 5x5, accessed by "arr[1][1]" → offset is 6

Similarly, the fields of the records will be held in contiguous locations and will be accessed by respective dereferences like : MyRecord.field1, MyRecord.field2 etc.

Lastly, your intermediate code will have one exception to the 4-address structure. For instructions for function calls, you will generate an instruction very similar to that in the source. Function calls with no return values will look like the following:

```
call, func_name, param1, param2, …, paramn
```

And function calls with return values will have a similar structure: callr,

x, func_name, param1, param2, …, paramn

The bodies of the function calls must be demarcated by adding a suitable annotation

#start_function <function_name> and #end_function <function_name>

For each correct program fed into your compiler, you should emit a complete instruction stream of intermediate code for that program. You do not need print any intermediate code for programs with errors.

2.D : Turn-in and grading

<u>Grading</u>

Deliverables for part 2:

1. Symbol table generation code that interfaces to ANTLR generated parser + Symbol Table	(20 points)
2. Semantic checking code	(35 points)
3. IR generation code + generated IR	(35 points)
4. Report (design internals, how to build, run examples etc.)	(10 points)

Appendix A: Tiger Language Reference Manual

Credit: Modified from Stephen A. Edwards' "Tiger Language Reference Manual" and from Appel's Modern Compiler Implementation in C

Grammar

<tiger-program> → main let <declaration-segment> in begin <stat-seq> end

<declaration-segment> → <type-declaration-list> <var-declaration-list> <funct-declaration-list>

<type-declaration-list> → NULL
<type-declaration-list> → <type-declaration> <type-declaration-list>

<var-declaration-list> → NULL
<var-declaration-list> → <var-declaration> <var-declaration-list>

<funct-declaration-list> → NULL
<funct-declaration-list> → <funct-declaration> <funct-declaration-list>

<type-declaration> → type id = <type> ;

<type> → <type-id>
<type> → array [INTLIT] of <type-id >
<type> → record <field-list> end
<type> → id

<field-list> → id : <type-id> ; <field-list>
<field-list> → NULL

<type-id> → int | float

<var-declaration> → var <id-list> : <type> <optional-init> ;
<id-list> → id
<id-list> → id, <id-list>
<optional-init> → NULL
<optional-init> → := <const>

<funct-declaration> → function id (<param-list>) <ret-type> begin <stat-seq> end ;
<param-list> → NULL
<param-list> → <param> <param-list-tail>
<param-list-tail> → NULL
<param-list-tail> → , <param> <param-list-tail>

```
 <ret-type> → NULL
 <ret-type> →  : <type>
<param> → id : <type>

<stat-seq> → <stat>
<stat-seq> → <stat> <stat-seq>

<stat> →  <lvalue>  <l-tail> :=  <expr> ;
<l-tail> → :=  <lvalue>  <l-tail>
<l-tail> → NULL
<stat> → if  <expr> then <stat-seq>  endif  ;
<stat> → if  <expr > then <stat-seq> else <stat-seq> endif;
<stat> → while  <expr> do <stat-seq> enddo;
<stat> → for id := <expr>  to <expr> do <stat-seq> enddo;
<stat> → <opt-prefix> id( <expr-list> ) ;
<opt-prefix> → <lvalue> :=
<opt-prefix> → NULL
<stat> → break;
<stat> → return<expr> ;

<stat> → let <declaration-segment> in <stat-seq> end

<expr> → <const>
         → <lvalue>
         →  <expr> <binary-operator> <expr>
         →  ( <expr> )
<const> → INTLIT
<const> → FLOATLIT

<binary-operator> → ** | + | - | * | / | = | <> | < | > | <= | >=| & |  |

<expr-list> →  NULL
<expr-list> →   <expr> <expr-list-tail>
<expr-list-tail> → , <expr> <expr-list-tail>
<expr-list-tail> → NULL

<lvalue> → id <lvalue-tail>
<lvalue-tail> → [ <expr> ]
<lvalue-tail> → . id
<lvalue-tail> → NULL
```

Lexical Rules
Identifier: sequence of one or more letters, digits, and underscores. Must start with a letter can be followed by zero or more of letter, digit or underscore. Case sensitive.
Comment: begins with /* and ends with */. Nesting is not allowed.
Integer constant: sequence of one or more digits. Should not have leading zeroes. Should be unsigned.
float constant: Must have at least one or more digits before the decimal points followed by a decimal point. There could be zero or more digits after the decimal point. Floats are also unsigned.

Reserved (key) words
```
main array record break do else end for function if in let of then to
type var while endif begin end enddo return
```

Punctuation Symbols
, : ; ( ) [ ] { }

Binary Operators
+ - * / ** = <> < > <= >= & |

Assignment operator
:=

Precedence (Highest to Lowest)
( ) ** * / + - = <> > < >= <= & |

Operators
Binary operators take integer or float operands and return an integer or a float result. Comparison operators compare their operands, which may be either both integer or both float and produce the integer value 1 if the comparison holds (indicating a true) and integer 0 otherwise (indicating a false). The binary operators = and <> can compare any two operands of the same type and return either integer 0 or 1. Integers are the same if they have the same value. The binary operators **, +, -, *, and / require two operands and return result; mixed expression between integers and floats is permitted, the result being a float for all except **. The exponentiation operator ** can have its left operand as integer or a float but the right operand must be an integer; it is a semantic error otherwise. The +, -, *, and / operators are left associative. On the other hand, ** is right associative, ie, a ** b ** c evaluates as b raised to c first (let the result be t) and then a is raised to t. The assignment operator is also right associative e.g. a := b := c; first evaluates c and assigns the value of c to that of b; this new value of b is then assigned to a. Integer value is auto-promoted to a float during the assignment but assignment of a float to integer is a type mismatch error. No aggregate operations are allowed, ie, all the operators must operate on scalar values (and not on arrays nor on records as aggregates).

Zero is considered false; one is considered true. Parentheses group expressions in the usual way. The comparison operators do not associate, e.g., a=b=c is erroneous. The logical operators & and | are logical "and" and "or" operators.   They take a logical and or of the conditional results and produce the combined result. Aggregate operation on arrays or records is illegal – they must be operated on an element by element or field by field basis.

Arrays

We have static arrays in Tiger. An array of any named type may be made by `array [intlit] of` *type-id* of length intlit. Arrays can be created only by first creating a type. The dereferencing of an array can be done by creating an index expression which must be only integer type. Example, A[2*i + j] – the array expression evaluates to l-value or r-value depending on where it appears, as an l-value – it evaluates to a storage in which we store a value, as an r-value it returns a value stored in that array location.

Records

Records are similarly declared by first creating a type and then using it to create a variable. The fields can of different base types (int or float) and no initialization is allowed on records. The fields of a record can only be scalars – arrays are not allowed inside records.

Types

Two named types `int` and `float` are predefined. Additional named types may be defined or redefined (including the predefined ones) by type declarations.

The two production rules for *type* (see the grammar rules above) refer to

1. a type (creates an alias in a declaration)
2. an array from a base type
3. a record consisting of several fields each created from a base type either int or float

Type equivalence enforced in the compiler is name type equivalence – ie, variables a and b defined to be of same named type are considered equivalent. In other words, even if two entities are structurally equivalent (such as two arrays or records of same lengths and corresponding field types), they will not be treated equivalent by the compiler.

In `let` . . . *type-declaration* . . . `in` *expr-seq?* `end`, the scope of the type declaration begins at the start of the sequence of type declarations to which it belongs (which may be a singleton) and ends at the end. Type names have their own name space.

Assignment

The assignment expression *lvalue* := *expr* evaluates the expression then binds its value to the contents of the *lvalue*. Assignment operator is right associative as noted earlier (see the section of operators). Aggregate assignment is illegal on arrays or records and must be done on an element by element or field by field basis.

Control Flow
The if-then-else expression, written `if` *expr* `then` *<stat-seq>* `else` *<stat-seq>* `endif` evaluates the first expression, which must return an integer. If the result is non-zero, the statements under the then clause are evaluated, otherwise the third part under else clause is evaluated.

The if-then expression, `if` *expr* `then` *<stat-seq>* `endif` evaluates its first expression, which must be an integer. If the result is non-zero, it evaluates the statements under then clause.

The while-do expression, `while` *expr* `do` *<stat-seq>* evaluates its first expression, which must return an integer. If it is non-zero, the body of the loop <stat-seq> evaluated, and the while-do expression is evaluated again.

The for expression, `for` *id* := *expr* `to` *expr* `do` *<stat-seq>* evaluates the first and second expressions, which are loop bounds. Then, for each integer value between the values of these two expressions (inclusive), the third part <stat-seq> is evaluated with the integer variable named by *id* bound to the loop index. This part is not executed if the loop's upper bound is less than the lower bound.

Let
The expression `let` *declaration-list* `in` *<stat-seq>* `end` evaluates the declarations, binding types, variables, and functions to the scope of the expression sequence, which is a sequence of zero or more semicolon-separated statements in <stat-seq>.

Let Scoping rules

Let statements can be nested as per the grammar. The scopes follow the classic block structure. Here's how the block structure works for Tiger. Each let statement opens a new scope which ends at the corresponding end of the let statement. The binding rules for let follow block structure as follows:

- A declaration of an entity (types, variables and functions) with a given name in a given scope hides its declaration from outer scopes (if any). For example, consider a declaration of a variable : TechStudent in inner scope, it will hide all the outer scope declarations of TechStudent if any.
- When a scope is closed by the corresponding end, all entities declared in that scope are automatically destroyed. That is, the the lifetime of entities is limited to the scope in which they are declared and when the scope is closed, the entity declared in that scope ceases to exist.
- Lookup rules: The binding of a name is decided in the following manner: first the name is looked up in the current scope, if the look-up finds the declared name in the current scope, it is bound to the corresponding entity. If not found, the lookup proceeds to the outer scopes one by one until the name is found. If a declaration of the name is not found anywhere, it means the entity is undeclared and constitutes a semantic error. Once a name is found in a given scope, the lookup stops and does not proceed to outer scopes. As an example,

consider scope 1 being the outermost scope and inner scopes being 2 and 3 respectively (innermost scope here is scope 3). Consider the lookup in scope 3. The lookup will work as follows: first the entity will be looked up in innermost scope 3, if found the name is bound to that declaration, if not the look-up proceeds to scope 2. If the entity is found in scope 2, the name will be bound to that declaration, if not the lookup continues to outermost scope 3 and the procedure is repeated. If all three lookups fail , it is a case of semantic error of undeclared entity.

Variables

A *variable-declaration* declares a new variable and its initial value (optional). Lifetime of a variable is limited to its scope. Variables, types and functions share the same name space. Redeclaration of the same name in the same scope is illegal.

Functions

The first form is a procedure declaration (no return type); the second is a function (with return type). Functions return a value of the specified type; procedures are only called for their side-effects. Both forms allow the specification of a list of zero or more typed arguments, which are passed by value.

Standard Library

function print(s : string)

Print the string on the standard output.

function printi(i : int)

Print the integer on the standard output.

function flush()

Flush the standard output buffer.

function getchar() : string

Read and return a character from standard input; return an empty string at end-of-file.

function ord(s : string) : int

Return the ASCII value of the first character of s, or −1 if s is empty.

function chr(i : int) : string

Return a single-character string for ASCII value i. Terminate program if i is out of range.

function size(s : string) : int

Return the number of characters in s.

function substring(s:string,f:int,n:int):string

Return the substring of s starting at the character f (first character is numbered zero) and going for n characters.

function concat (s1:string, s2:string):string

Return a new string consisting of s1 followed by s2.

function not(i : int) : int

Return 1 if i is zero, 0 otherwise.

function exit(i : int)

Terminate execution of the program with code i.

Sample Tiger Programs (Scalar Dot Product)

Example I:

```
main
 let
   type ArrayInt = array [100] of int;  /*Declare ArrayInt as a new type */
   type MyRecordType = record sum : int; product: int ; end; /* Declare MyRecordType as a new
type*//
    var  X, Y : ArrayInt = 10;  /*Declare vars X and Y as arrays with initialization */
    var i : int = 0;
    var dotproduct : MyRecordType;
in
 begin
    dotproduct.sum := 0;
    dotproduct. product :=1;
    for i := 1 to 100 do                          /* for loop for dot product */
          dotproduct.sum := dotproduct.sum + X[i] * Y[i];
          dotproduct. product := dotproduct.product * X[i] * Y[i];
     enddo
     printi(dotproduct.sum);   /* library call to printi to print the dot product */
    printi(dotproduct.product); /* print the product */
end
```

Appendix B: IR for Tiger

Assignment: (op, x, y,_)

Op	Example source	Example IR
assign	a := b	assign, a, b,

Binary operation: (op, y, z, x)

Op	Example source	Example IR
add	a + b	add, a, b, t1
sub	a - b	sub, a, b, t1

mult	a * b	mult, a, b, t1
div	a / b	div, a, b, t1
and	a & b	and, a, b, t1
or	a | b	or, a, b, t1

Goto: (op, label, _, _)

Op	Example source	Example IR
goto	break;	goto, after_loop, ,

Branch: (op, y, z, label)

Op	Example source	Example IR
breq	if(a <> b) then	breq, a, b, after_if_part
brneq	if(a = b) then	brneq, a, b, after_if_part
brlt	if(a >= b) then	brlt, a, b, after_if_part
brgt	if(a <= b) then	brgt, a, b, after_if_part
brgeq	if(a < b) then	brgeq, a, b, after_if_part
brleq	if(a > b) then	brleq, a, b, after_if_part

Return: (op, x, _, _)

Op	Example source	Example IR
return	return a;	return, a, ,

Function call (no return value): (op, func_name, param1, param2, …, paramn)

Op	Example source	Example IR
call	foo(x);	call, foo, x

Function call (with return value): (op, x, func_name, param1, param2, …, paramn)

Op	Example source	Example IR
callr	a := foo(x, y, z);	callr, a, foo, x, y, z

Store into array: (op, array_name, offset, x)

Op	Example source	Example IR
array_store	arr[0] := a	array_store, arr, 0, a

Load from array: (op, x, array_name, offset)

Op	Example source	Example IR
array_load	a := arr[0]	array_load, a, arr, 0

Array Assignment: (op, x, size, value)

Op	Example source	Example IR
assign	var X : ArrayInt := 10; /* ArrayInt is an int array of size 100 */	assign, X, 100, 10