The course project is to build a compiler for a small language. This is a “living” document will be revised throughout the semester until it is a complete, if sometimes informal, language specification. Revisions may include additions, removals and changes to meet pedagogical goals and to ensure internal consistency.

( X ) means zero or one occurrence of X
{ X } + means one or more occurrences of X
{ X } *  means zero or more occurrences of X

SECTION 1: Lexical structure – see milestone 1 document (version 1.0)
SECTION 2: Syntactic structure – see milestone 2 document (version 1.5)

SECTION 3: Type checking and semantics

Add the –tc compiler option, to report type errors in the ASC file. If the -tc flag is not specified no type checking is done, and no type errors are reported. Add actions to the rules of your grammar to perform type checking, insert type conversions where required (see notes associated with grammar rules), and otherwise report type errors when they occur. Type checking must occur as appropriate, including (but not necessarily limited to) these places:

In the following, the expression must be of type Boolean:

```
for (' statement ';' expression ';' statement ')' sblock
while (' expression ')' sblock
if (' expression ')' then sblock else sblock
! expression
```

In the following, the expression must be of type integer, and the constants must be of type integer:

```
switch (' expression ')' { case constant ':' sblock }+ otherwise ':' sblock
i2r expression
```

In the following, the expression must be of type real:

```
r2i expression
```

In the following, the expression or assignable must be a type allocated on the heap. Records and arrays are allocated space on the heap. Nothing else is explicitly allocated space on the heap. ‘reserve’ allocates space on the heap. ‘release’ frees space previously allocated on the heap. In ‘reserve’, if the assignable is an array, the size of each dimension must be given, as in reserve arr(7,4), which reserves space for a 7 by 4 array of elements according to the declaration of arr.

```
expression isNull
reserve assignable
release assignable
```

In the following, exp1 and exp2 must be of the same type (no coercion). exp1 must be assignable. If the assignment occurs inside a function body and exp1 is the name of the function, then the type of exp1 and exp2 must be the same as the return type of the function (the effect is that of a return statement in C or Java).

```
exp1 := exp2
```
In the following, \( \text{exp1} \) must be a record type:

\[
\text{exp1} \cdot \text{exp2}
\]

except in the special case where \( \text{exp1} \) is an array type and \( \text{exp2} \) is of the form \( _0, _1, \text{etc} \) (as described in the milestone 2 document for the rule \textit{assignable is assignable recOp identifier} (pages 3 and 4)).

In the following, the expression must be either integer or real:

- \( -\text{expression} \)
- \( +\text{expression} \)

In the following, \( \text{exp1} \) and \( \text{exp2} \) must be either both integer or both real, or (if one is integer and the other real) a type conversion operation from integer to real must be inserted by the compiler:

\[
\text{exp1} + \text{exp2} \\
\text{exp1} - \text{exp2} \\
\text{exp1} \times \text{exp2} \\
\text{exp1} / \text{exp2}
\]

In the following, \( \text{exp1} \) and \( \text{exp2} \) can be of any of the types integer, real, Boolean, or character, as long as \( \text{exp1} \) and \( \text{exp2} \) have the same type, or (if one is integer and the other real) a type conversion operation from integer to real must be inserted by the compiler:

\[
\text{exp1} < \text{exp2}
\]

In the following, \( \text{exp1} \) and \( \text{exp2} \) can be of any type, under the following constraints: (1) either \( \text{exp1} \) and \( \text{exp2} \) have the same type, or (2) if one is an integer and the other real, then a type conversion operation from integer to real must be inserted by the compiler, or (3) if one is the constant null, then the other may be of an array type, a record type, or a function type:

\[
\text{exp1} = \text{exp2}
\]

In the following, \( \text{exp1} \) and \( \text{exp2} \) must be both be integer:

\[
\text{exp1} \% \text{exp2}
\]

In the following, \( \text{exp1} \) and \( \text{exp2} \) must be both be Boolean:

\[
\text{exp1} \& \text{exp2} \\
\text{exp1} \mid \text{exp2}
\]

In the following, if assignable refers to a function, then the number, type and order of expressions in ablock must be identical to that given in the function’s domain type (see discussion below for one special case). If, on the other hand, assignable refers to an array, then ablock must have the number of integer expressions given by the constant in the array’s type.

\text{assignable ablock}
CHECKPOINT – Section 3 should be completed by Friday, March 31. This is a guideline: there is no separate submission due on March 31.

SECTION 4: Intermediate code generation

Add the –ir compiler option, to produce the intermediate representation of a program to a file with the extension ‘.ir’. In the output produced use symbolic labels (regardless of what your compiler-internal representation is).

Use the intermediate representation instructions given in section 6.2.1 of the text, on pages 364-365. Your team may choose whichever internal representation it wishes.

Review 6.3.4 – 6.3.6. Generate intermediate code for programs processed by your compiler, under the following assumptions:

- integer – 32-bit wide two’s complement
- real – 64-bit wide IEEE 754
- character – 8-bit wide ASCII
- Boolean – 8-bit wide

Array – fixed size, determined by initial allocation. The number of dimensions is determined by type declaration and is known in the symbol table at compile time. For each dimension there is a 4-byte block storing an integer denoting the size (number of elements) of that dimension. Your team must decide whether to use row-major or column major order. Arrays are zero-indexed (lowest index is always 0). See 6.4.3 – 6.4.4.

String – a one-dimensional array of character. In other words, of a fixed size, determined by initial allocation. The first 4-byte block stores the size (number of characters) as an integer. Elements of string (values of type character) are stored in consecutive bytes. String literals are a shorthand way of creating an array of characters.

Record – fixed size, determined by sizes of its constituent elements and alignment requirements.

Assume the size of a pointer is 64 bits. Arrays and records are allocated in the heap using reserve, and are therefore accessed indirectly via a pointer.

Generate code for int->real type coercions. Generate code as if the proper explicit type conversion had been present in the source code (e.g. i2r exp rather than just exp), and only where the grammar allows it (i.e. NEITHER in explicit nor implicit assignments – implicit assignments occur in function calls). See 6.5.2 – 6.5.3. The ONLY overloading allowed in our language is with the arithmetic operators (+, -, *, and /), and <.

Assume our binary Boolean operators are short-circuiting. Generate code for flow-of-control statements (for, while, if-then-else, and switch). See 6.6 – 6.8. The semantics for flow-of-control statements is typical (we will review in class).
Generate code for bounds-check array access. We will discuss in detail how arrays are laid out in memory, but the size of each dimension of an array is stored as part of its in-memory representation and can be used to ensure that each array access uses an in-range index.

Generate code for function definitions and function calls as outlined in section 6.9. Note that every function technically takes exactly one argument and returns exactly one value. In addition to the normal function definition syntax we will support a special syntax for a function whose domain type is a record with more than one member, to give the illusion of a function of multiple parameters.

For example,

```plaintext
type rec: [integer: x, y]
type T1: integer -> integer
type T2: rec -> integer

function foo : T1
function bar1 : T2
function bar2 : T2

foo(x) := {
    return x * x;
}

bar1(a) := {
    return a.x * a.y;
}

bar2(r,s) := {
    return r * s;
}

entry(arg) := {
    [ integer: result ; rec: w]
    result := foo(5);
    w := reserve(w);  (* see types.alpha – reserve returns a value of type address,
                       which can be assigned to array and record variables *)
    w.x := 5;
    w.y := 7;
    result := bar1(w);  (* pass w (a rec type value) to bar1 *)
    result := bar2(5,7);  (* implicitly build a rec type value, assign 5 and 7 to fields x and y, but call them r and s *)

    return 0;
}
```

Standard operators have expected semantics:

unary: -, !
binary: +, -, *, /, %, &|, !, <, =, :=  (use ‘==’ as the three address code translation of ‘=’,
                                          and ‘=’ as the three address code translation of ‘:=’)

Special operators: assume that they are defined:
Unary: isNull, reserve, release, i2r, r2i

(translate \( exp \) isNull as \( exp == \) null; translate reserve(x) as \( x = malloc(y) \), where y is the size of x, in bytes; release(x) as free(x); the rest as themselves)

Functionality from previous stages
Your submission should not only provide the functionality of this stage, but also that described in previous project stages.

SUBMISSION & GRADING:
Submit your code using Autolab. Submissions are due no later than 5:00 PM on Friday April 14.