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 version 1.1 of this document)
SECTION 2: Syntactic structure (see version 2.0 of this document)

SECTION 3: Type checking and semantics

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.

> 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.

> exp1 := exp2

In the following, exp1 must be a record type:

> exp1 . exp2

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

- expression

In the following, exp1 and 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:

exp1 + exp2
exp1 – exp2
exp1 * exp2
exp1 / exp2

In the following, exp1 and exp2 can be of any of the types integer, real, Boolean, or character, as long as exp1 and 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:

exp1 < exp2

In the following, exp1 and exp2 can be of any type, under the following constraints: (1) either exp1 and 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:

exp1 = exp2

In the following, exp1 and exp2 must be both be integer:

exp1 % exp2

In the following, exp1 and exp2 must be both be Boolean:

exp1 & exp2
exp1 | 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 pblock of the function's type. 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.

assignable ablock

CHECKPOINT – Section 3 should be completed by Wednesday, March 28.

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; first 1 byte of array stores the number of dimensions for the array type, then for each dimension there is a 4-byte block storing an int denoting the size (number of elements) of that dimension. Use row-major order. Arrays are zero-indexed (lowest index is always 0). See 6.4.3 – 6.4.4.
>
> String – an array of character. In other words, of a fixed size, determined by initial allocation; first byte holds 1 (since a String is a one-dimensional array), and then the next 4 bytes of store 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.

Generate code for int->float 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 /.

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 function definitions and function calls as outlined in section 6.9.

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 malloc(y), where y is the size of x, in bytes; release(x) as release(z,s), where z is the address of x and s is the size of 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 Monday April 9.