The α programming language

Carl Alphonce
Revised 2024-04-01

This document describes the α programming language, a high-level programming language which serves as the basis for the compiler project in the University at Buffalo course *CSE443 Compilers*.

Toolchain requirements

1. Tools:
    a. `flex: /usr/bin/flex (2.6.4)`
    b. `bison: /util/bin/bison (3.8.2)`
    c. `C: /usr/bin/gcc (9.4.0)`
2. Execution environment:
    a. `cerf.cse.buffalo.edu`
    b. `turing.cse.buffalo.edu`

Toolchain recommendations

3. Editor
    a. `emacs (29.1 or later)`

The description of the α programming language has undergone significant changes for spring 2024 to streamline the language. While the changes have been carefully considered and reviewed it is possible that further clarifications are needed. Please don't hesitate to ask on Piazza if you find something incongruous in this document.

SECTION 1: Lexical structure

The lexical structure of the language is defined (informally) below.

Parentheses are used for grouping, the pipe '|' for alternation. For example, ('e' | 'E') means either the lower case letter 'e' or the upper case letter 'E' (but not both).

'?' indicates optionality (zero or one occurrence). For example, ('+'|'-')? means either the plus sign '+' or the minus sign '-' can appear, but neither is required.

'+' indicates one or more occurrence. For example, digit+ means one or more digits (where digit is defined as '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')

Section 1.1: Specifications

Legal identifiers must begin with an upper or lower case letter or '_', followed by an arbitrarily long string of upper or lower case letters, '_', and digits.

The language has five built-in types, the first four primitive and the last composite. Literal values for each type are described. Note that numeric literals cannot be negative.

`integer` – 32-bit wide two's complement numbers: `digit+`

`address` – a 64-bit wide memory address or `null`. `null` is the only literal value.

`Boolean` – the two values `true` and `false`

`character` – 8-bit wide 7-bit ASCII characters: literals are characters in single quotes, e.g. 'a' or one of the following '\'-escaped characters: '\n' (newline), '\t' (tab), '\'' (single quote), and '\\' (backslash).

`string` – a sequence of values of type character, of arbitrary length, enclosed in double quotes, but not spanning more than one line. The double-quote character must be '\'-escaped in string literals, whereas the single-quote character is not, as shown in this example:

"I'd think this is a legal \"string\" that contains \n \t several escaped characters, isn't it?"

<u>Keywords (all keywords are *reserved*)</u>:

the names of the primitive types (given above), and

```
true
false
null
while
if
then
else
type
function
return
external
as
```

<u>Punctuation</u>:

Parentheses: `(` `)`
Brackets: `[` `]`
Braces: `{` `}`
Other punctuation: `;` `:` `,` `->` `"` `\`

<u>Operators</u>:

Arithmetic operators: `+` `-` `*` `/` `%`
Relational operators: `<` `=`
Assignment operator: `:=`
Logical operators: `!` `&` `|`
Member access (dot) operator: `.`

<u>Whitespace and comments:</u>

space, tab, newline
comments are delimited by `(*` and `*)`

Section 1.2: Standard token constants

To standardize the output from every team's lexer, use the following constants to represent your tokens:

```
// identifier
ID          101

// type names
T_INTEGER   201
T_ADDRESS   202
T_BOOLEAN   203
T_CHARACTER 204
T_STRING    205

// constants (literals)
C_INTEGER   301
C_NULL      302
C_CHARACTER 303
C_STRING    304
C_TRUE      305
C_FALSE     306

// other keywords
WHILE       401
IF          402
THEN        403
ELSE        404
TYPE        405
FUNCTION    406
RETURN      407
EXTERNAL    408
AS          409

// punctuation – grouping
L_PAREN     501
R_PAREN     502
L_BRACKET   503
R_BRACKET   504
L_BRACE     505
R_BRACE     506

// punctuation – other
SEMI_COLON  507
COLON       508
COMMA       509
ARROW       510

// operators
ADD         601
SUB_OR_NEG  602
MUL         603
DIV         604
REM         605
LESS_THAN   606
EQUAL_TO    607
ASSIGN      608
NOT         609
AND         610
OR          611
DOT         612
RESERVE     613
RELEASE     614
```

SECTION 2: Syntactic structure

The language is defined (informally) as follows; part of your job is to define a reasonable formal grammar that you can use with Flex and Bison to parse/compile programs written in the language. **Keywords** appear in bold.

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

Section 2.1: Productions

program is:

prototype-or-definition-list — *A function named `entry` must be defined to generate an executable (see sample program below).*

prototype-or-definition-list is:

prototype prototype-or-definition-list — *At least one prototype or definition is required. In order to create an executable file the function **entry** must be defined.*

definition prototype-or-definition-list

prototype

definition

prototype is: — *Function prototype (i.e. a function declaration)*

(**external**) **function** $identifier_1$ ':' $identifier_2$ — *identifier1 is function name, identifier2 is function type **external** communicates to the compiler that this function is user-defined, but is defined in another file rather than this one. It will be compiled separately and must be linked in to produce a valid executable.*

definition is: — *Defines a new type or function*

type identifier ':' dblock — *Record type. identifier is name of the record type*

type $identifier_1$ ':' constant '->' $identifier_2$ — *Mapping (array) type. identifier1 is name of the array type constant: an integer, the number of dimensions (note: not the SIZE of those dimensions) identifier2 is name of element type*

type $identifier_1$ ':' $identifier_2$ '->' $identifier_3$ — *Mapping (function) type. identifier1 is name of the function type*

identifier2 is name of the domain type
Identifier3 is name of the range type

identifier parameter assignOp sblock

Function definition

identifier is function name

parameter is the name of the parameter, possibly elaborated by an 'as' clause

sblock is function body

parameter is:

'(' identifier ')'

as '(' idlist ')'

Every function has exactly one parameter. To name that parameter enclose an identifier in parentheses. If the parameter is a record type names can be associated with the record elements using an ***as*** *clause.*

idlist is:

identifier ',' idlist

identifier

sblock is:

'{' (dblock) statement-list '}'

sblock allows local declarations in optional dblock Scope of the sblock starts at the '{'. This determines the scope number for the symbol table output.

dblock is:

'[' declaration-list ']'

declaration-list is:

declaration ';' declaration-list

declaration

declaration is:

identifier ':' identifier

LHS is type, RHS is a variable name

statement-list is:

compound-statement statement-list

while(…) { } x := y ;

compound-statement	*while(…) { }*
simple-statement ';' statement-list	*x := y ; while(…) { }*
simple-statement ';'	*x := y ; or return exp ;*

compound-statement is:	*Ends with a '}'*
while '(' expression ')' sblock	*Boolean expression*
if '(' expression ')' **then** sblock **else** sblock	*Boolean expression, else is required*
sblock	*Nested block & therefore nested scope*

simple-statement is:	*Does not end with '}'*
assignable assignOp expression	
return expression	

assignable is:

identifier	*Variable (could be name of function)*
assignable ablock	*Function call or array access.*

Can be assigned to only as an array access.

Size of ablock must match number of array dimensions (for array access) or number of parameters (for function call – more discussion below).

For array access each member of ablock must be an integer, an in-bounds array index.

For function call each member of ablock must be of the correct type, as determined by function's domain type.

Recall that technically every function has exactly one parameter. However, if the parameter is of a record type and the function was defined with the ***as*** *clause then a call with multiple apparent arguments is permitted. In this case the (implicit) record will be represented on the stack rather than indirectly on the heap.*

When used as an r-value the type is the assignable's range type

Pass-by-value, as in Java (i.e. value could be an address).

assignable recOp identifier	*Record access, or array dimension lookup*
	Accessing the size of each array dimension: if a is an n-dimensional array, allow a to be used in a record access construct as well: a._1 through a._n give access to the sizes of each of the n dimensions. Taking a concrete example, if foo is a 3-dimensional array of character and we reserve foo(5,4,7), then foo._1 has value 5, foo._2 has value 4, and foo._3 has value 7. The size of each dimension is determined dynamically (at runtime).
	a._0 denotes the number of dimensions of the array. Thus, foo._0 is 3. This is determined at compile time by the definition of the array's type.
expression is:	*Recall: an expression has a value.*
constant	*Literal, e.g. 17,* **`true`**, **`false`**, **`null`**, *"foo".*
UnaryOperator expression	
assignable	
expression binaryOperator expression	
'(' expression ')'	*Parenthesized expression.*
memOp assignable	*Value is pointer to memory block, or* **`null`**
ablock is:	
'(' argument-list ')'	*ablock must have parentheses.*
argument-list is:	
expression ',' argument-list	
expression	
UnaryOperator is:	
-	*Unary numeric negative.*
!	*Logical negation.*
memOp is:	
reserve	*Allocates space for type object.*

release — *Releases space for type object.*

> *ASIDE: RESERVE & RELEASE IN EXPLICIT & IMPLICIT ASSIGNMENTS*
> *Suppose 'arr' refers to a one-dimensional array of records. arr := reserve arr(10) reserves space for an array with 10 elements; the value assigned to arr is a pointer to the allocated block of memory. arr(1) := reserve arr(1) reserves space for a record; the value assigned to arr(1) is a pointer to the allocated block of memory.*
>
> *How do we determine whether reserve arr(x) allocates space for an array with x elements or for the type of value that can be stored in location arr(x)? In other words, how can the compiler determine whether x refers to a quantity or an index? The disambiguation comes from the expected type imposed by the LHS of the assignment. In the first case we are assigning to arr (a pointer to an array), in the latter to arr(x) (a pointer to an array element, in this case a record). The types of these expressions are different, and must be propagated as the 'expected type' for the RHS expression.*
>
> *The same applies in a function call: f(reserve arr(x)). Recall that there is an implicit assignment from each argument in a function call to the corresponding parameter in the parameter list.*
>
> *Assume that 'release exp' always returns the null pointer. There is no ambiguity with release similar to that for reserve. release arr would release the memory pointed to by arr (i.e. the memory occupied by the array), whereas release arr(1) would release the memory pointed to by arr(1), under the assumption of course that arr(1) was a pointer type.*

assignOp is:

:= — *Assignment.*

recOp is:

. — *Record access.*

binaryOperator is: — *Usual prec/assoc rules apply.*

+

-

*

/

%

& — *Logical AND, short circuiting.*

| — *Logical OR, short circuiting.*

< — *Relational operators: less than*

defined for numeric types: *i*i->b, c*c->b (numeric '<')*
defined for Boolean: *b*b->b (false<true)*

= *Relational operator: equal to*
*Defined for all types t: t*t->b*

Section 2.2: Precedence/Associativity table (from highest precedence to lowest precedence)

OPERATOR	DESCRIPTION	ASSOCIATIVITY
reserve, release	Memory allocation	N/A (unary)
.	Record access	N/A (unary)
-	Unary	N/A (unary)
!	Logical negation	N/A (unary)
*, /, %	Binary	left-to-right
+, -	Binary	left-to-right
<		left-to-right
=	Equality	left-to-right
&		left-to-right
|		left-to-right
:=	Assignment	N/A (cannot be part of expression)

Parenthesized expressions have highest priority.

SECTION 3: Type checking and semantics

Type checking must occur as appropriate, including (but not necessarily limited to) the following constructions.

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

> **while** '(' expression ')' sblock
> **if** '(' expression ')' **then** sblock **else** sblock
> ! 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.

> reserve assignable
> release assignable

In the following, exp1 and exp2 must be of the same type. 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, exp1 must be a record type:

> exp1 . exp2

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

In the following, the expression must be integer:

> - expression

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

> exp1 + exp2
> exp1 – exp2
> exp1 * exp2
> exp1 / exp2
> exp1 % exp2

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

> 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 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 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 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.

assignable ablock

SECTION 4: Intermediate code generation

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

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.

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,

```
type rec: [integer: x; integer: 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 as (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: reserve, release

SECTION 5: Assembly code generation

The compiler must generate *x86-64* assembly language instructions that preserve the semantics of the original source program. We will use a restricted subset of the available ISA. Information about the subset as well as general x86-64 resources are provided on the course website.

The expectation is that your compiler will:

a) generate assembly code that preserves semantics of source code program,
b) perform appropriate register allocation and assignment, and
c) possibly perform some simple optimizations, depending on what we have time to cover in lecture.

You must develop a test suite of programs to verify the correctness of your code generation.

APPENDIX A: The α library

The α programming language has a small library of functions that interface with functions in the C libraries. These functions provide very basic printing and memory allocation/freeing capabilities:

```
external function printInteger: integer2integer
external function printCharacter: character2integer
external function printBoolean: Boolean2integer

external function reserve: integer2address
external function release: address2integer
```

These functions, along with the standard entry function,

```
function entry: string2integer
```

are declared, and several useful types are defined, in a file named `library.alpha`.

This file should be included in any alpha code file by using the C macro preprocessor. The C macro preprocessor (`cpp`, also invoked as `gcc -E`) will process `#include` directives in α source code files. For more information on `cpp` see https://gcc.gnu.org/onlinedocs/cpp/

Before compiling an alpha file with a #include you must first run the preprocessor. Here is an example of how to run the preprocessor on a file named `simple.alpha`:

```
cpp -P -x c -o simple.cpp.alpha simple.alpha
```

This command write the output of the preprocessor to the file `simple.cpp.alpha`, which will contain pure alpha code.

Example

Suppose `simple.alpha` contains

```
#include "library.alpha"

entry(arg) := {
    return 0;
}
```

and `library.alpha` contains

```
(* At compiler start-up your program should create symbol table entries for the four primitive types:
       Boolean    (1 byte)
       character  (1 byte)
       integer    (4 bytes)
       address    (8 bytes)
   You should #include this file at the start of your alpha file.
   Some useful types are defined below.
*)

type string: 1 -> character

type BooleanXBoolean: [Boolean: x, y]
type characterXcharacter: [character: x, y]
type integerXinteger: [integer: x, y]
```

```
type Boolean2Boolean: Boolean -> Boolean
type integer2integer: integer -> integer

type character2integer: character -> integer
type Boolean2integer: Boolean -> integer
type string2integer: string -> integer

type integerXinteger2integer: integerXinteger -> integer
type integerXinteger2Boolean: integerXinteger -> Boolean
type characterXcharacter2Boolean: characterXcharacter -> Boolean
type BooleanXBoolean2Boolean: BooleanXBoolean -> Boolean


type integer2address: integer -> address
type address2integer: address -> integer

external function printInteger: integer2integer
external function printCharacter: character2integer
external function printBoolean: Boolean2integer
external function reserve: integer2address
external function release: address2integer

function entry: string2integer
```

After running the preprocessor with the command shown above the file `simple.cpp.alpha` will be created with the following contents:

```
(* At compiler start-up your program should create symbol table entries for the four primitive types:
       Boolean    (1 byte)
       character  (1 byte)
       integer    (4 bytes)
       address    (8 bytes)
   You should #include this file at the start of your alpha file.
   Some useful types are defined below.
*)
type string: 1 -> character
type BooleanXBoolean: [Boolean: x, y]
type characterXcharacter: [character: x, y]
type integerXinteger: [integer: x, y]
type Boolean2Boolean: Boolean -> Boolean
type integer2integer: integer -> integer
type character2integer: character -> integer
type Boolean2integer: Boolean -> integer
type string2integer: string -> integer
type integerXinteger2integer: integerXinteger -> integer
type integerXinteger2Boolean: integerXinteger -> Boolean
type characterXcharacter2Boolean: characterXcharacter -> Boolean
type BooleanXBoolean2Boolean: BooleanXBoolean -> Boolean
type integer2address: integer -> address
type address2integer: address -> integer
external function printInteger: integer2integer
external function printCharacter: character2integer
external function printBoolean: Boolean2integer
external function reserve: integer2address
external function release: address2integer
function entry: string2integer
entry(arg) := {
    return 0;
}
```

APPENDIX B: Compiler invocation

Section B.1: Making the compiler

Put all necessary code into a zip file. Include a makefile named 'Makefile', which has at least two targets: compiler and clean. To make the current up-to-date compiler:

```
make compiler
```

To remove any files generated by `make compiler`, invoke `make clean`

Section B.2: Command-line Arguments

Your executable must accept the command-line arguments indicated in the output below, which must itself be produced when the compiler is invoked with the 'help' command-line argument (where '`%`' is the OS prompt and '`./alpha -help`' is the compiler invocation):

```
% ./alpha -help
HELP:
How to run the alpha compiler:
./alpha [options] program
Valid options:
-tok   output the token number, token, line number, and column number for each of the tokens to the .tok file
-st    output the symbol table for the program to the .st file
-asc   output the annotated source code for the program to the .asc file, including syntax errors
-tc    run the type checker and report type errors to the .asc file
-ir    run the intermediate representation generator, writing output to the .ir file
-cg    run the (x86 assembly) code generator, writing output to the .s file
-debug produce debugging messages to stderr
-help  print this message and exit the alpha compiler
```

For example, suppose that `prog1.alpha` is an input file, then running

```
% ./alpha -tok prog1.alpha
```

must produce output in a file named `prog1.tok`

Similarly, if `prog2.alpha` is an input file, then running

```
% ./alpha -tok prog2.alpha
```

must produce output in a file named `prog2.tok`

Invoking with other command-line flags must trigger the indicated behavior. For example,

```
./alpha -tok -st -asc prog.alpha
```

must produce the token, Annotated Source Code, and Symbol Table files for **prog**. The order of flags is irrelelvant, so the above invocation is equivalent to this:

```
./alpha -asc -st -tok prog.alpha
```

Section B.3: The compiler flags in detail

Assume **alpha** is the name of your compiler, and that **prog.alpha** contains :

```
(* Type definitions *)
type string: 1 -> character
type int2int: integer -> integer
type string2int: string -> integer

(* Function prototypes
   They use the above type definitions
*)
function square : int2int
function entry : string2int

(* Function definition
   Functions must be declared before they are defined
*)
square(x) := {
    return x * x;
}

(* Function definition
   entry is the first function called
*)
entry(arg) := {
  [ integer: input; integer: expected; integer: actual; boolean: result; string: input ]
  input = 7;
  expected = 49;
  actual := square(input);
  rseult := expected = actual;
  return 0;
}
```

Section B.3.1: the -tok option / token stream

The lexer component of your compiler must write (to a file as indicated below) the numeric value representing each token in the input file, a space, the text that matched the token, a space, the starting line number of the token, a space, the starting column number of the token, followed by a new line character, using code along these lines:

```
fprintf(FILE_tok, "%d %d %3d \"%s\"\n", lineNumber, columnNumber, token, text)
```

Your lexer must also recognize and print to the output file but not return to the parser, the following pseudo-token for comments:

```
// comments
COMMENT     700
```

Section B.3.2: the -st option / symbol table

Invoking

```
./alpha -st prog
```

must lex and parse the contents of **prog** and produce a **s**ymbol **t**able description to the file **prog.st**

The symbol table must be written to the file in the following format (this example does not show all pre-loaded types):

```
NAME             : SCOPE  : PARENT : TYPE                 : Extra annotation
-----------------:--------:--------:----------------------:-----------------------------
Boolean          : 001001 :        : primitive            : type
character        : 001001 :        : primitive            : type
integer          : 001001 :        : primitive            : type
string           : 001001 :        : 1 -> character       : type
int2int          : 001001 :        : integer -> integer   : type
string2int       : 001001 :        : string -> integer    : type
square           : 001001 :        : int2int              : function
entry            : 001001 :        : string2int           : function
-----------------:--------:--------:----------------------:-----------------------------
x                : 014014 : 001001 : integer              : parameter (of square)
-----------------:--------:--------:----------------------:-----------------------------
arg              : 021015 : 001001 : string               : parameter (of entry)
input            : 021015 : 001001 : integer              : local
expected         : 021015 : 001001 : integer              : local
actual           : 021015 : 001001 : integer              : local
result           : 021015 : 001001 : $_undefined_type     : local
-----------------:--------:--------:----------------------:-----------------------------
```

The 'extra annotation' indicates the kind of declaration, along with other useful information.

Each scope is identified by the line number and column number where it begins. The global scope is always 001001. Each scope aside from scope 001001 has a parent scope (indicated in the PARENT column above).

The extra annotation 'type' means that the identifier being introduced (such as string2int) is the name of a type. string -> integer is a function type, mapping a string to an integer.

The extra annotation 'parameter (of *name*)' means that the identifier being introduced (such as x) is the name of a parameter for the named function.

The extra annotation ‘function’ means that the identifier being introduced (such as square) is the name of a function.

The extra annotation ‘local’ means that the identifier being introduced (such as input) is the name of a local variable.

`$_undefined_type` is a compiler-internal type name used as the type of any expression with an undefined type. Note that as it starts with ‘$’ it cannot conflict with any user-defined type.

Section B.3.3: the -asc option / annotated source code

Invoking

```
./alpha -asc prog.alpha
```

should lex and parse the contents of **prog** and produce annotated source code to the file **prog.asc**

In this case the source code listing contained in **prog.asc** should be:

```
001: (* Type definitions *)
002: type int2int: integer -> integer
003: type string2int: string -> integer
004:
005: (* Function prototypes
006:    They use the above type definitions
007: *)
008: function square : int2int
009: function entry : string2int
010:
011: (* Function definition
012:    Functions must be declared before they are defined
013: *)
014: square(x) := {
015:     return x * x;
016: }
017:
018: (* Function definition
019:    entry is the first function called
020: *)
021: entry(arg) := {
022:   [ integer: input; integer: expected; integer: actual; boolean: result; string: input ]
LINE 022:51 ** ERROR: the name 'boolean', used here as a type, has not been declared at this point in the program.
LINE 022:60 ** ERROR: the name 'result' is not declared with a valid type
LINE 022:74 ** ERROR: the name 'input' has already declared at this point in the program
023:   actual := square(input);
024:   rseult := expected = actual;
LINE 024:3 ** ERROR: the name 'rseult', used here as a variable name, has not been declared at this point in the program.
025:   return 0;
026: }
```

In this example type checking is not turned on.

Each line begins with a zero-padded three-digit line number, a colon, and a space.

Error messages should all begin with “`LINE` *lineNumber*`:`*columnNumber* `** ERROR:`”, and then give a description of what the error was. The description of the error need not be exactly as shown (you should come up with messages that are as meaningful as you can make them, without being overly wordy).

Your parser must produce error messages for errors identified by the LALR parse table, as well as undeclared names identified by symbol table lookup, and type errors. There may be other errors that your parser identifies, in which case they should be included in the parser output as well. The above is not intended to be a definite statement of the parser's output, but an indication of the format expected. You may use the standard *syntax error* messages that Bison produces, in addition to the *name error* messages shown in the above sample output.

Section B.3.4: the -tc option / type checking

Invoking

```
./alpha -asc -tc prog.alpha
```

must report type errors in the ASC file. Invoking without the `-asc` flag would still perform type checking, but type error would not be reported. 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 and report type errors when they occur. You must craft meaningful *type error* messages.

Section B.3.5: the -ir option / intermediate representation

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

Invoking

```
./alpha -ir prog.alpha
```

must write a representation of the compiler's internal intermediate representation of a program the file **prog.ir**

Section B.3.6: the -cg option / code generation

Add the –cg compiler option, to produce x86-64 assembly code of a program to a file with the extension '.s'.

Invoking

```
./alpha -cg prog.alpha
```

must write a representation of the x86-64 assembly representation of a program the file **prog.s**

Section B.3.7: the -debug option

Add the –debug compiler option. This option is intended to produce development-time debugging messages inserted by and the for the use of the development team. Without this option specified NO extraneous output may by produced.

Section B.3.8: the -help option

The following output must be produced when the compiler is invoked with the 'help' command-line argument (where '`%`' is the OS prompt and '`./alpha -help`' is the compiler invocation):

```
% ./alpha -help
HELP:
How to run the alpha compiler:
./alpha [options] program
Valid options:
-tok   output the token number, token, line number, and column number for each of the tokens to the .tok file
-st    output the symbol table for the program to the .st file
-asc   output the annotated source code for the program to the .asc file, including syntax errors
-tc    run the type checker and report type errors to the .asc file
-ir    run the intermediate representation generator, writing output to the .ir file
-cg    run the (x86 assembly) code generator, writing output to the .s file
-debug produce debugging messages to stderr
-help  print this message and exit the alpha compiler
```

APPENDIX C: ADDITIONAL RESOURCES

Lexical Analysis with Flex (2.6.0)
https://epaperpress.com/lexandyacc/download/flex.pdf

Bison (3.8.1)
https://www.gnu.org/software/bison/manual/html_node/index.html

Though this refers to lex and yacc (rather than flex and bison) you might find the following on-line tutorial helpful:

https://www.epaperpress.com/lexandyacc/download/LexAndYacc.pdf

Using as
https://sourceware.org/binutils/docs/as/

Stanford CS107 *Guide to x86-64*
https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/guide/x86-64.html

Stanford CS107 *Handy one-page of x86-64*
https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/onepage_x86-64.pdf

X86-64 Register and Instruction Quick Start
https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start

ASCII
https://en.wikipedia.org/wiki/ASCII

Matt Godbolt's compiler explorer site
https://godbolt.org

Intel 64 and IA-32 Architectures Software Develop's Manual (VERY LARGE)
https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4

System V Application Binary Interface AMD64 Architecture Processor Supplement
https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf