

- The zlang page has language details, nuances, and caveats you might be interested in.
- You can download all the resources used in this writeup: zobos-resources.zip or zobos-resources.tar.bz2.
The tarball contains source files (`.src`, `.syn`), token streams (`.tok`), AST PDFs, and verbose and OUTPUT filtered `stdout` listings (`.vrb` and `.out`) respectively. **Read the README.txt!**
- Here is the change log for this assignment write-up. I will try to be descriptive in my log messages.
- You can also subscribe to this page and receive Emails when changes are made.

In this assignment will you will implement a syntax checking SLR(1) parser, a symbol table(s), and perform semantic analysis on a simple programming language grammar. The name of your application must be ZOBOS (case sensitive).

Input

ZOBOS will accept three command line parameters:

Argument	Value	Description
1	<code>program.tok</code>	The filename containing a token stream to be parsed, the format is identical to that produced by LUTHER
2	<code>ast.dat</code>	The filename for storing your AST
3	<code>symtable.dat</code>	The filename for storing symbol tables when instructed.

Coding

ZOBOS will do do the following:

1. Your compiler will parse a token stream (the output of LUTHER) with the LR knitting needles algorithm (pseudo code here), using this SLR table (the format is described here) and the zlang.cfg grammar. Syntax error messages will be produced in a consistent manner to facilitate grading.

- If program source has a syntax error, the remaining steps for ZOBOS are not to be performed and ZOBOS should `exit()` with a non-zero exit status.

2. Either during the parse (using **syntax directed translation** procedures during production rule reduction, show `_lr-knitting-with-sdt.pdf`, `lga-sdt.pdf`), after the parse using **visitor patterns**, or a combination of these methods; ZOBOS will construct an *abstract syntax tree* with the following properties
 - a. all EXPR (sub)trees will be simplified so that
 - i. leaves are either literals (`intvals`, `floatvals`) or variables (identifiers holding a value, not function names)
 - ii. the root and internal nodes are either one of the operations associated with the non-terminals `BOOLS`, `PLUS`, `MULT`, `UNARY`, or `CAST`, or a `FUNCALL` node.
 - b. control structures `IF`, `IFELSE`, `WHILE`, `STMTS`, and `BRACESTMTS` will be simplified **in the spirit** of textbook figure 7.15 (`lga-sdt.pdf`) --- essentially lose the terminals, keep the functional children (blocks of statements, predicate expressions).

Simplification of all other grammar constructs are at the student's discretion and inclination. They will not be graded.

There are example `.src` and AST PDFs in the `zobos-resources` archives for download.

Outside of the rules listed above, **you are not required** to duplicate the ASTs provided, they are provided as examples!

3. ZOBOS will write to disk a simple tree representation of its AST.
4. ZOBOS will then use a scope respecting symbol table and visitor pattern(s) to perform several semantic checks on the variable, expression and function use within the parsed program source.
 - a. When the special `EMIT` form of `symtable` is encountered during this process, your ZOBOS will write the symbol tables for all scopes to disk in a simple format to facilitate grading.
 - b. Warnings or error messages will be emitted in a consistent and simple manner to facilitate grading.
5. Lastly, ZOBOS will `exit(0)` if there were **no errors** detected in the program, and `exit(1)` otherwise.

WARNING messages are not ERROR messages.

The ZOBOS language

The language compiled for this project is "C-like", but with some caveats and nuances. See this page for details.

You may either hard code the LR parsing table and grammar rules into your ZOBOS application or read `zlang.lr` and the grammar rules from disk on each invocation. **If the latter**, include `zlang.lr` and (or) `zlang-pure.cfg` in your submitted archive along side `Build.sh` or the `Makefile`; which will probably be where ever your ZOBOS is built. See this page to recall submission details.

Syntax and Semantic Checks

Your ZOBOS compiler will inspect the `program.tok` for these types of errors and inconsistencies.

Keep in mind that we often clump all errors into the *syntax error*, pigeonhole. Technically, **syntax errors** occur only during the parse when an invalid pattern of terminals is detected. The other errors that compilers find for us, and *most of them* in this project, are **semantic** errors and inconsistencies.

Our language does not have "first class" functions, and functions won't be treated as identifiers holding a "value". The language has data types `bool`, `int`, `float` and `string`; and then it has functions. Think C/C++, not Python, JavaScript, Scheme or Haskell.

Type	ID	Description	Terminal Location to report (line and column)
SYNTAX	SYNTAX	Any syntax error during the parse	Token causing error, or last token successfully processed
ERROR	CALL	Invoking a variable identifier as a function, using the wrong number of arguments in a function call	Location of the function identifier being invoked
WARN	UNINIT	Using an identifier in an expression before it has been initialized with a value	Location of identifier in expression
WARN	REIDENT	Attempting to re-declare an identifier	Each occurrence of identifier in re-declaration

There are several other types of semantic checks could be performed, so many in fact that only a few are

used for each ZOBOS project per semester. You might read elsewhere in the write-up conditional statements such as *"If required by the semester write-up, such-and-such would be a FOOBAR warning."* Depending on the semester these statements are either important or can be safely ignored.

The table above is the authoritative list of warnings and errors your ZOBOS should report; use it to guide your implementation.

Symbol Table Contents and Functions as (sort of a) Type

While functions are not first class objects in our language, we will adopt semantics around function prototypes and definitions so that ERROR and WARN messages, as well as the symbol table contents don't have to be specialized for function types. A symbol table should have the following (minimum) attributes per entry for ZOBOS success:

- location (line and column reported for the identifier token, see here for function definitions and prototypes),
- identifier (name),
- type (see here for function "types"),
- a `const` flag,
- a "used" or "unused" flag,
- an "initialized" or "uninitialized" flag.

When an identifier for a function is encountered, its value in the symbol table depends on the source context:

1. When a function prototype is encountered, the symbol is considered **not const**, but **already initialized**.
2. When a function definition is encountered the symbol has its `const` flag set (**or** is created with its `const` and `initialized` flag true).

These semantics mean your ZOBOS does not have to have different logic for processing variables versus functions (aside from the initial management of function symbols just described). The lion's share of function specific code producing output is for CALL error messages.

OUTPUT

ZOBOS will report three types of data to three different files.

Syntax, Warning and Error Messages

These will be written to system `stdout` and **must be** prefixed by the token `OUTPUT` as described by the course submission requirements. These messages will have four critical parts emitted in this order:

1. the type of message `:TYPE:`, enclosed with colons¹ where `TYPE` is one of (all capitals) `SYNTAX`, `ERROR`, or `WARN`.
2. the line number of the error in the source (see *Terminal Location* column of syntax, error, and warning table)
3. the character number of the error in the source (see *Terminal Location* column of syntax, error, and warning table)
4. the ID for the message type

The line and character number come from the `program.tok token stream`. For a `SYNTAX` error it is the location of the offending token *or* the the last good token *iff* the syntax error occurs because there are no more tokens to be parsed. For `WARN` and `ERROR` messages, the location to report is detailed here.

Example Syntax Error Output

Since we abort all tasks on a syntax error, there should only be one `SYNTAX` error ever printed on `stdout`. You **are permitted to print other information**, just don't put it on the required `OUTPUT` line:

`danglingbrace.vrb` (the source `danglingbrace.syn` is available in the ZOBOS archive files)

```
OUTPUT :SYNTAX: 2 14 :SYNTAX:
Parsing error in state 64; input: ['sc', ';'], ['lbrace', '{'], ['if', 'if']; expected=lbrace
```

You aren't required to print out anything beyond the data on the `OUTPUT` line.

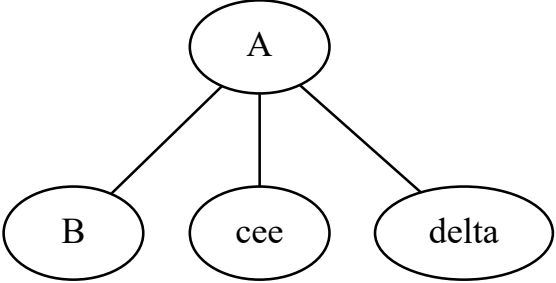
Abstract Syntax Tree

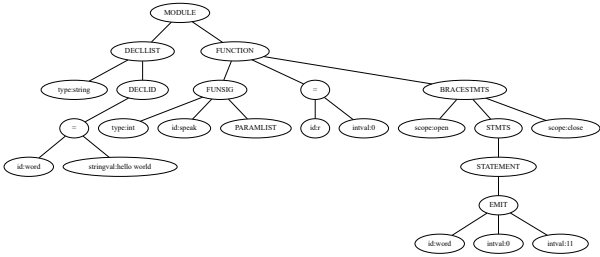
You have the option of writing a `dot(1)` instruction file **instead of** the format described below. The `grader.sh` script will auto-detect the format you use.

ZOBOS will write its abstract syntax tree to the filename specified by the second command line parameter. The format of this file is in two parts. The **first part** is a line by line declaration of a tree node **id** and **name**, separated by white space. If a name is not provided the **id** is used.

An **empty line** separates the the first part from the second part.

The **second part** is a line by line description of tree edges. The first id on the line is the parent, subsequent ids are the children in left to right order. If a parent id is mentioned more than once, children are simply arranged in left to right, file order. The following two AST descriptions in this format yield the same tree graph:

First AST Description	Second AST Description	dot() Generated Graph
<pre> A A B C cee D delta A B C D </pre>	<pre> A A B C cee D delta A B A C D </pre>	 <pre> graph TD A((A)) --- B((B)) A --- cee((cee)) A --- delta((delta)) </pre>

Source (helloworld.src)	AST Description (helloworld.ast)	dot() Generated Graph
<pre> string word = "hello world"; int speak() returns r = 0 { emit word, 0, 11; } </pre>	<pre> 0 MODULE 0-0 DECLLIST 0-0-0 type:string 0-0-1 DECLID 0-0-1-0 = 0-0-1-0-0 id:word 0-0-1-0-1 stringval:hello world 0-1 FUNCTION 0-1-0 FUNSIG 0-1-0-0 type:int 0-1-0-1 id:speak 0-1-0-2 PARAMLIST 0-1-1 = 0-1-1-0 id:r 0-1-1-1 intval:0 0-1-2 BRACESTMTS 0-1-2-0 scope:open 0-1-2-1 STMTS </pre>	 <pre> graph TD MODULE --> DECLLIST MODULE --> FUNCTION DECLLIST --> type_string["type:string"] DECLLIST --> DECLID DECLID --> eq["="] DECLID --> stringval["stringval:hello world"] eq --> id_word["id:word"] FUNCTION --> FUNSIG FUNCTION --> eq2["="] FUNCTION --> BRACESTMTS FUNSIG --> type_int["type:int"] FUNSIG --> id_speak["id:speak"] FUNSIG --> PARAMLIST eq2 --> id_r["id:r"] eq2 --> intval_0["intval:0"] BRACESTMTS --> scope_open["scope:open"] BRACESTMTS --> STMTS BRACESTMTS --> scope_close["scope:close"] STMTS --> STATEMENT STATEMENT --> EMIT EMIT --> id_word2["id:word"] EMIT --> intval_02["intval:0"] EMIT --> intval_11["intval:11"] </pre>

```

0-1-2-1-0 STATEMENT
0-1-2-1-0-0 EMIT
0-1-2-1-0-0-0 id:word
0-1-2-1-0-0-1 intval:0
0-1-2-1-0-0-2 intval:11
0-1-2-2 scope:close

0-0-1-0 0-0-1-0-0 0-0-1-0-1
0-0-1 0-0-1-0
0-0 0-0-0 0-0-1
0-1-0 0-1-0-0 0-1-0-1 0-1-0-2
0-1-1 0-1-1-0 0-1-1-1
0-1-2-1-0-0 0-1-2-1-0-0-0
0-1-2-1-0-0-1 0-1-2-1-0-0-2
0-1-2-1-0 0-1-2-1-0-0
0-1-2-1 0-1-2-1-0
0-1-2 0-1-2-0 0-1-2-1 0-1-2-2
0-1 0-1-0 0-1-1 0-1-2
0 0-0 0-1

```

The example above suggests an easy way to create unique identifiers for all AST nodes: the digit patterns are simply the path to each node, and each path element is the child index.

Symbol Tables

During symbol table generation and the search for semantic warnings and errors, if ZOBOS encounters the `emit symtable` statement all the symbols for all scopes should be written to the **third command line argument**. The format for this output is as follows:

1. The **global** or deepest scope will be considered scope zero (0), each nested scope within it is assigned the next sequential scope value.
2. Each symbol should be written on one line, and contain the following **comma delimited fields**: scope value, type, and id.
 - a. The scope value is a non-negative integer, 0 is used for the deepest or "global" scope.
 - b. **type** is the sequence of grammar terminals associated with a variable's DECLTYPE or it is the **function signature**:

```
returnType// (param1type (/param2type (/param3type (/...))) )
```

eg, the function signature for `speak` in the `helloworld.src` example above is simply `int//`.

c. The `id` is the identifier of the symbol.

1. Yes, it is possible to have scopes without any symbols declared within them.
2. Within the same scope, if an identifier is attempted to be re-used,
 - a. If required by the semester write-up a REIDENT error should be generated
 - b. Regardless of error generation, the attempted re-declaration **is ignored**.

All of the `.src` files with `emit symtable` instructions in the zobos resource archives have `.sym` files provided as well. Here is an example of program source with multiple scoping levels and the `emit symtable` output.

Source (symtable-2.src)	Symbol Table Dump (symtable-2.sym)
<pre> string m = "helloworld"; const float pi = 3.1415; int i = 0; const bool b = 1<3; int main() returns r=0 { string p = m; int f = i; string i = "eyespyeashadow"; { { bool c = b; emit p, 0, int(b); if (f > 9.81) { c = bool(f*f-int(c)); emit symtable; } emit i, 0, 15; } emit m, int(pi), 5; } } </pre>	<pre> 0,const bool,b 0,int,i 0,const string,m 0,const int//,main 0,const float,pi 1,int,r 2,int,f 2,const string,i 2,const string,p 4,bool,c </pre>

Hints and Testing

1. You know you're going to dump symbol table debug info during development, so you might as well format it according to the requirements in the first place.

grader.sh

I am providing to students the same tarball the grader will use for testing your ZOBOS. Here is how to use it:

First, download this tarball to your Mines Linux account ("alamode" machines!) and unroll it in a temporary directory.

```
$ ls zobos-student.*
zobos-student.tar.bz2
$ mkdir ~/tmp
$ cd ~/tmp
$ tar xjf ../zobos-student.tar.bz2
:
$ ls
zobos/
```

Second, set the COMPGRADING environmental variable with:

```
$ source ~khellman/COMPGRADING/setup.sh ~khellman/COMPGRADING
```

Now go to the directory holding your ZOBOS and execute the `grader.sh` script from the `zobos-student.tar.bz2` resource.

```
$ cd ~/compilers/zobosCC
$ ls ZOBOS
ZOBOS
$ ~/tmp/zobos/grader.sh
:
:
:
```

You will need to read any messages from the script carefully, and perhaps hit ENTER several times throughout its course. This script checks for:

- a. missing data files
- b. truncated data files
- c. inaccessible data files
- d. and the difference between your ZOBOS results and expected results.

The latter test results are displayed on the terminal screen, but they whip by pretty quickly so you may need to scroll up and make sure you see them all.

When there is a discrepancy in expected results the script either:

- points you to a data file along side your ZOBOS where more failure details are available,
- or all the details are displayed on screen.

Before the `grader.sh` terminates, it will show a summary of some specific rubric line results for the programming project.

The grader will use this very same script, possibly with some additional testing `.src` to check your submitted work. If your ZOBOS flies through without a hitch, you can likely suspect a good grade for the assignment.

Submit Your Work

Partners

You may complete this assignment in a group of two or three other students from the course (you may also work solo if you choose). Exams may have questions specific to course programming assignments — so be sure your group is truly working as a team and have a solid understanding of your submission's design and algorithms.

If you decide to complete this assignment as a group, you **must** tell your instructor **one week before** the assignment due date. **Only one of you should** submit the assignment for grading. You will all be able to see the grading result from your own logins.

Rubric

This work is worth 100 points.

Requirements	Points	Notes
Meets compilers course project requirements	10	
Basic I/O and exit status requirements	10	exit status non-zero on unreadable input files or un-writable output files

Detect and report SYNTAX errors correctly	20	exit status should be non-zero when SYNTAX errors are detected
AST simplification of "raw" parse tree, output of AST graph information	20	
emit <code>symtable</code> and <code>symtable.dat</code> requirements	20	
ERROR class semantic issues properly detected and reported	10	exit status should be non-zero when ERROR issues are detected; 75% correct and no extra messages for full credit
WARN class semantic issues properly detected and reported	10	exit status should be zero when only WARN issues are detected; 75% correct and no extra messages for full credit

1. the course submission requirements again. (1)

Assignments/ZOBOS (last edited 2022-04-26 09:19:01 by khellman)