Please follow the policies for assignments given on the course website. The assignment must be turned in electronically: Send a gzip-compressed tar-ball to cs328-submit@bloat.org. Include all the necessary files to build and run your project on ugradx.cs.jhu.edu, but do not include any derived files; we expect to build your project using make; we expect to run your project using ./sc. The name of the archive should be cs328-assign-5-login.tar.gz where login is replaced with your login name on the ugradx.cs.jhu.edu server.

You must put your name, login, and contact email into every file you turn in. All documents you turn in must be in PDF format, generated by pdflatex; please turn in your LATEX source files as well.

You are free to use the “standard library” for your language of choice (C++, Java, Python), except for modules that allow you to avoid writing the code for the assignment (no regular expression functionality, no parsing frameworks, etc.). You will be graded on functionality (70%), packaging (10%), design (10%), and documentation (again 10%).

## 1 The Abstract Syntax Tree (90%)

In the fourth part of the compiler project, you will extend your parser to build an abstract syntax tree (AST) for the instructions, expressions, and conditions in a SIMPLE program. You will also enforce several context conditions that have not been checked so far. These tasks are part of the semantic analysis for SIMPLE. The concrete grammar for SIMPLE is given in EBNF on page 6, the context conditions you must enforce are given on page 7, and the abstract grammar for SIMPLE—together with some helpful comments—is given on page 8.

### 1.1 Driver Program

First you need to adapt your driver program to integrate the abstract syntax tree. With this assignment, the option -a is allowed on the command line for sc. The remaining options, including “no options given,” should still result in errors, except for -s, -c, and -t of course, which are unchanged.

The option -a is supposed to build and display the abstract syntax tree for a given input program. If -a is given but an error is detected, the (partial) abstract syntax tree should not be printed. Since the semantic actions to build the abstract syntax tree and to enforce context conditions will be inside the parser, you need to be able to turn them off in case only -c or -t is given. Consider adding a bool parameter to the parser’s constructor for this, but other solutions are possible as well.

Also, you need to decide how you will access the abstract syntax tree once the parser is done building it. One option is to add a method tree, which returns the abstract syntax tree to the driver for printing, to the parser. But you’re pretty much on your own... There’s not much more to say about the driver, a more detailed description of what it has to produce for this assignment is given below.

### 1.2 Abstract Syntax Tree

The abstract syntax tree will keep track of the instructions, expressions, and conditions in a SIMPLE program. In fact, the AST will be the primary data structure for the interpreter and code-generator you will develop in the following assignments.

Nodes. First you need to decide how you will represent the instructions, expressions, and conditions in a SIMPLE program. The abstract grammar for SIMPLE de-
scribes the **structure** of the AST, but it doesn’t define the necessary details. I suggest you introduce a base class `Node` for all kinds of nodes in the AST; remember that you might want to introduce several abstract methods here later on. Derived from `Node` you should introduce classes `Instruction`, `Expression`, and `Condition` to model the “big three” categories.

Consider the abstract grammar for conditions for a moment. Obviously you can already make a concrete class in this case, storing two expression pointers (left and right) and the actual relation being checked (\(=\), \(!=\), \(<\), \(\leq\), or \(\geq\)). This is the “pattern” you will follow for the other concrete classes as well: each class will store what it is supposed to store according to its production in the abstract grammar.

Now consider the abstract grammar for instructions. As you can see, there are five different instructions that can occur in a SIMPLE program, so you will define classes `Assign`, `If`, `Repeat`, `Read`, and `Write` derived from the base class `Instruction`.

For expressions you can proceed in a similar fashion. For locations, things get a little tricky. However, I don’t want to give you too many hints (enjoy figuring it out yourself :-). Figure 1 illustrates what a complex designator like

\[
a[1].x[2].y := -20
\]

should be translated into; it also shows that the AST nodes `Number` and `Variable` actually refer to ST entries.

For purposes of type-checking you should give AST nodes derived from `Expression` a member variable that stores the type associated with it (for simplicity, we ignore the fact that the type of a field in an array is another array). This will allow you to check the type of a complex designator like

\[
a[1].x[2].y := -20
\]

and printing them; all the bugs you find here will not distract you later on...

**Building the AST.** The next step is to extend the parser methods for instructions, expressions, and conditions to create the proper AST nodes and subtrees. Each method should return the “top node” of the subtree it creates.

For example, in the method `Condition`, you first call `Expression`. This will parse the required number of tokens and return a pointer to an `Expression` node representing what it just recognized. Then you match one of the allowed relations and remember it. Then you call `Expression` again, getting a pointer to another `Expression` node, this time for the right-hand side of the comparison. Now you have all the “ingredients” to create a `Condition` node, filling in the left and right subtrees as well as the relation being checked. The method `Condition` then returns a pointer to the node it just created, to be used by whatever method called `Condition` in the first place (the parser method `Repeat` for example). Thus the AST is built “bottom-up” as you recognize the program text from left to right.

In this way, each of these methods returns the subtree...
it recognized, and its caller can “hook” that subtree into a larger one it in turn passes back. The tree for the complete program will be returned by the call to Instructions within Program. Note that in the case of Selector you will not just return a tree, but you also pass a tree as a parameter. Otherwise Selector would not know what it is being applied to, and you could not enforce the necessary context conditions. You should test your extensions with a number of simple example programs before you move on.

**Constant Folding.** In the methods parsing expressions you should now add **constant folding** as described in the lecture. Obviously, only **literal numbers** and identifiers that refer to symbol table **constants** are indeed constant. In both of these cases, you can return a **Number** node with a pointer to a **Constant** object. Before you produce **Binary** nodes, however, you should check whether both sides are constant. If they are indeed, you perform the operation **directly** and return a **Number** node with a pointer to a **Constant** filled with the result of the operation. Thus an expression with only **constant** parts will in fact **not** lead to a tree, but just to a **Number** node with the final value already computed.

Once constant folding works, you can “hook up” the AST with the ST from the previous assignment. Where you **assumed** the value 5 before, for example in **Type** when parsing an **ARRAY** constructor, you can now actually use the result of **Expression**. For array types, you will ensure that the expression you get back is indeed a **Number** node, that its value is greater than 0, and then fill in that value in your **Array** object. Note that this approach to constant folding is **far** from perfect. For example, the expression 1+a+3 where a is a variable will **not** be transformed into a+4: we only fold **adjacent** nodes and not whole expressions. However, it is good enough for our purposes in **SIMPLE**.

**Error Messages.** Enforcing context conditions for **SIMPLE** programs will lead to a number of “new” errors which you should handle by throwing exceptions as before. The **required** format is still as follows:

```plaintext
error: some helpful description
```

Since all that matters for automated grading is that you actually **detect** errors, your actual error messages would not have to make any sense. However, you should **try** to make your messages as informative as possible, for example by including details about **where** an error occurred. In other words, position information from the tokens should somehow “make it” into the nodes of your abstract syntax tree and into the error messages you produce. A simple way to do this is to remember the start of the first token and the end of the last token the parser matched before creating a certain AST node. When you create a node out of other nodes, you can use the minimum of all start positions and the maximum of all end positions for a rough approximation. For example, if come across the designator a[1].x[2].y, but the last record type does not actually have a field y, you could output the following (on one line of course):

```plaintext
error: the designator "a[1].x[2]" @ (128, 136) does not refer to a record with a field "y" @ (138, 138)
```

The details of how to do this are up to you, and the example is certainly not the best possible message. Note that “fancy” error messages are **not** required, but highly encouraged.

### 1.3 Output

Once you have successfully built the abstract syntax tree for an input program, you must produce output that illustrates its structure. Your starting point for the output should be the root of the AST you built for the program itself, which will output the entire program. The following examples are based on the **SIMPLE** program shown in figure 2.

The textual output for the program in figure 2 is shown in figure 3. As you can see, at certain places you will...
have to include output for pieces of the symbol table as well, hopefully your solution for the previous assignment is modular enough to handle this. I again suggest that you use the VISITOR design pattern [1] to traverse the abstract syntax tree once it is built. Inside your visitor you can maintain an indentation count as well as a string buffer in which you assemble the output. You can then print from your driver after a successful parse. You must have textual output in this form to get points for the assignment.

2 Graphical ASTs (10\%)

The graphical output for the program in figure 2 is shown in figure 4. Note that the output in graphical form is 100\% accurate, an (almost) exact representation of what the data structures should look like in memory. I once again recommend using the DOT tool for visualization.\(^1\) Note that you simply write the DOT format to the standard output. We will take care of actually rendering the graphics if we want to look at your tree while grading.

As in the previous assignment, the driver program must accept an option \(-g\) to indicate that \(-a\) should produce graphical output instead of textual output.\(^2\) Please follow the shapes used in figure 4 and use rectangles for all Node instances in the AST. You don’t have to include the complete symbol table output inside the AST output, things would get too complicated; instead, just indicate variables and constants as shown, don’t follow type pointers any further.

3 Graduate Option

If you are taking this course at the graduate level, you must include accurate position information in your fancy error messages just like in the previous assignments.

This is an additional requirement to get the full 90\% for Problem 1; making the graphical output work for fancy error handling falls under the 10\% for Problem 2.

4 Personal Log

Update your log from the previous assignment with information about what you worked on for the course since then. Note the “update” above, do not throw away what you handed in previously, just add another section. There

---


2. Just to make sure: Giving \(-a\) alone generates the basic textual output, giving \(-a \ -g\) generates the DOT output.
Figure 4: Example of graphical AST output.

is no set amount of text you must add, but we will grade on content and effort; flimsy logs don’t fly.

Note that the personal log is a participation grade item, not an assignment grade item.

References

5 Concrete Grammar for SIMPLE

5.1 Context-Free Part

Program = "PROGRAM" identifier ";" Declarations ["BEGIN" Instructions] "END" identifier "." .

Declarations = { ConstDecl | TypeDecl | VarDecl } .
ConstDecl = "CONST" {identifier "+" Expression ";"} .
TypeDecl = "TYPE" {identifier "+" Type ";"} .
VarDecl = "VAR" {IdentifierList "+" Type ";"} .

Type = identifier | "ARRAY" Expression "OF" Type | "RECORD" {IdentifierList "+" Type ";"} "END" .

Expression = ["+"]"-" Term ("+"|"-"|"-"|"-") Term .
Term = Factor ("+"|"-"|"DIV"|"MOD") Factor .
Factor = integer | Designator | "(" Expression ")" .

Instructions = Instruction (";" Instruction) .
Instruction = Assign | If | Repeat | While | Read | Write .
Assign = Designator "+=" Expression .
If = "IF" Condition "THEN" Instructions ["ELSE" Instructions] "END" .
Repeat = "REPEAT" Instructions "UNTIL" Condition "END" .
While = "WHILE" Condition "DO" Instructions "END" .
Condition = Expression ("="|"#"|"<"|">">"|"<="|">=") Expression .
Write = "WRITE" Expression .
Read = "READ" Designator .

Designator = identifier Selector .
Selector = ("[" ExpressionList "]" | "." identifier) .
IdentifierList = identifier ("," identifier) .
ExpressionList = Expression ("," Expression) .

5.2 Regular Part

identifier = letter {letter | digit} .
integer = digit {digit} .
letter = "a" | "b" | .. | "z" | "A" | "B" | .. | "Z" .
digit = "0" | "1" | .. | "9" .
6  Context Conditions for SIMPLE

1. All identifiers must be declared before they can be used; “before” means “textually preceding” in SIMPLE.

2. No identifier can be declared more than once in a given scope; however, an identifier from an enclosing scope can be shadowed in the current scope.

3. The identifier used after PROGRAM and the corresponding END must be identical; this identifier is not added to the symbol table.

4. The “lonely” identifier in the Type production (i.e. the first alternative) must denote a type; it is an error if the identifier denotes a constant or a variable.

5. The expression in the Type production must be constant, of type integer, and greater than zero.

6. The expression in a constant declaration must be constant, i.e. we must be able to evaluate it (fold it completely) at compile time. The type of the expression must be integer; there are no array or record constants.

7. Arithmetic operators are only applicable to operands of type integer, i.e. you cannot apply them to arrays or records.

8. The Designator in the Factor production must denote a variable or a constant, i.e. it cannot denote a type.

9. The Designator in the Assign production must denote a variable. The type of the left-hand side must be compatible with the type of the right-hand side under occurrence equivalence (aka “name equivalence”), i.e. they must refer to the same type entry in the symbol table. Note that you can assign arrays and records.

10. The expressions in the Condition production must both be of type integer, i.e. you cannot compare arrays or records.

11. The expression in the Write production must be of type integer, i.e. you cannot output arrays or records.

12. The designator in the Read production must denote a variable of type integer, i.e. you cannot input arrays or records.

13. The identifier in the Designator production must denote a variable or a constant, i.e. it cannot denote a type.

14. Selectors “[]” are only applicable to variables of array type; the type resulting from a single selector application is the element type of the array type. The type of each expression in the ExpressionList must be integer. Selectors “.” are only applicable to variables of record type; the type resulting from a single selector application is the field type of the denoted field.
7 Annotated Abstract Grammar for SIMPLE

Parts in bold indicate where nodes of the AST point to usable symbol table (ST) objects, namely constants and variables. Note that ST objects are not necessarily declared explicitly, e.g. for literal numbers. Nodes that require type information to enforce context conditions should also point to the relevant types in the symbol table. Again, ST types are not necessarily only those with an explicit declaration, e.g. for selector cascades.

Instructions = Instruction | Instruction Instructions.

Models a list of at least one instruction; the order of instructions in the input program matters and must be preserved.

Instruction = Assign | If | Repeat | Read | Write.

There are five kinds of instructions: assignment, if, repeat, read, and write. There are no while instructions! Each while in the input program is transformed into a repeat nested inside an if instead.

Assign = Location Expression.
Read = Location.

The destination of an assignment or read must be a writable location in memory, which can be one of three things:

Location = Variable | Index | Field.
Index = Location Expression.
Field = Location Variable.

A location is either a variable, an element of an array variable indexed by an expression (think of “lots of anonymous” variables), or a field within a record variable, which is itself a variable. The production for fields might be confusing at first: The point is that we need to keep track of the record variable that was selected from as well as the field variable that was selected within the record; but whereas the record part can be produced by preceding selectors, the variable part cannot (see concrete grammar).

If = Condition Instructions_{true} [Instructions_{false}].
Repeat = Condition Instructions.
Write = Expression.

The instructions for the “else” part of an if can be empty.

Expression = Number | Location | Binary.
Binary = Operator Expression_{left} Expression_{right}.

Inside expressions, locations are not used as destinations but as sources of values. The operator in a binary expression can be +, −, *, DIV, or MOD.

Condition = Relation Expression_{left} Expression_{right}.

The relation in a condition can be =, #, <, >, <=, or >=.