Course Project

Step 4

Code Generation

Project steps

1. Step 0 –
   ANTLR setup (~1 week)
2. Step 1 - Scanner
   (~2 weeks)
3. Step 2 - Parser
   (~3 weeks)
4. Step 3 - Symbol Table
   (~3 weeks)
5. Step 4 - Semantic actions and Code generation
   (~4 weeks)

Part 1: Expressions
Part 2: Control Structures

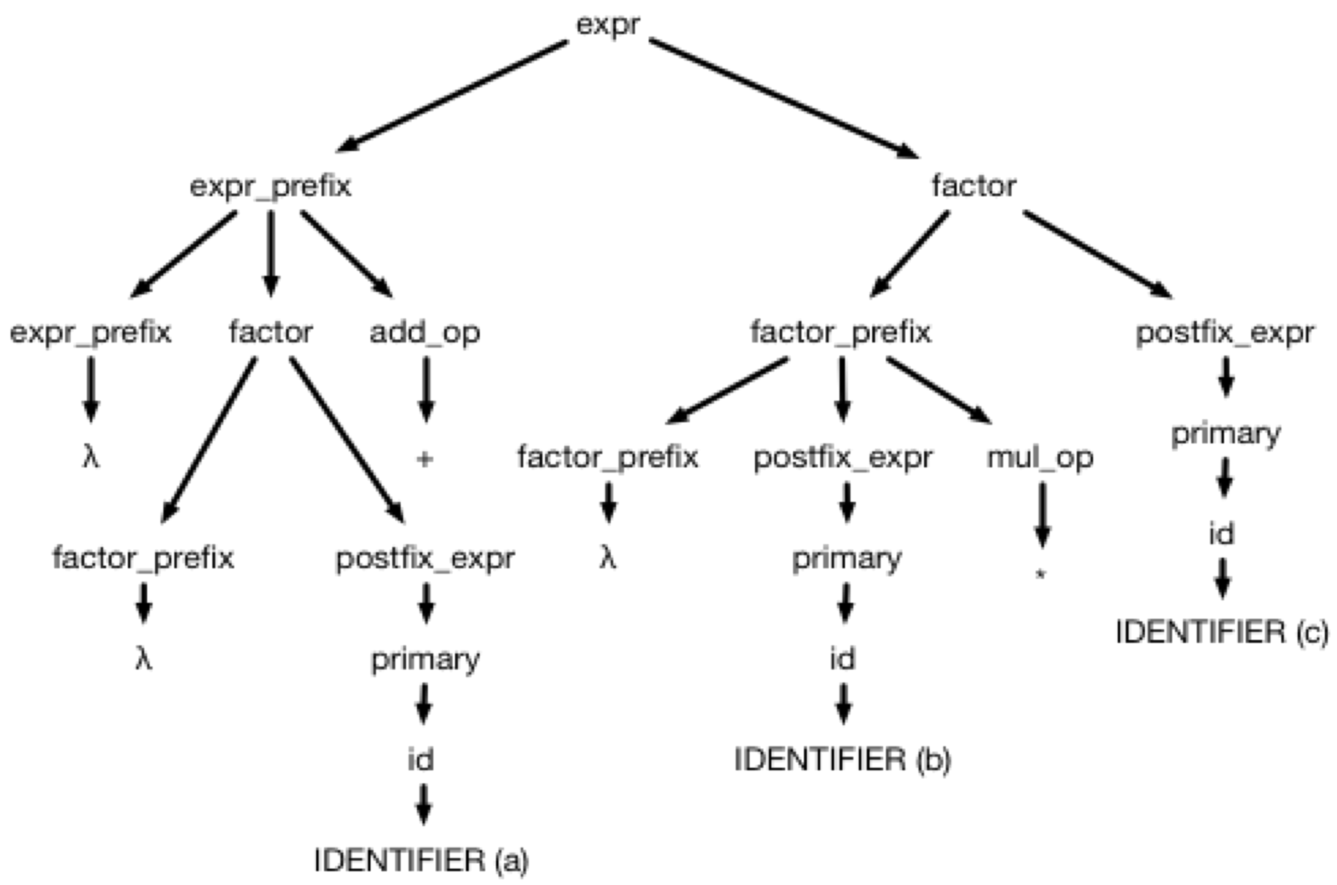
Project Step 4 — Part 1: Expressions

1. Generate an *abstract syntax tree* (AST) for the code in your function.
2. Convert the AST into a sequence of *IR Nodes* that implement your function using three address code.
3. Traverse your sequence of IR Nodes to generate assembly code.

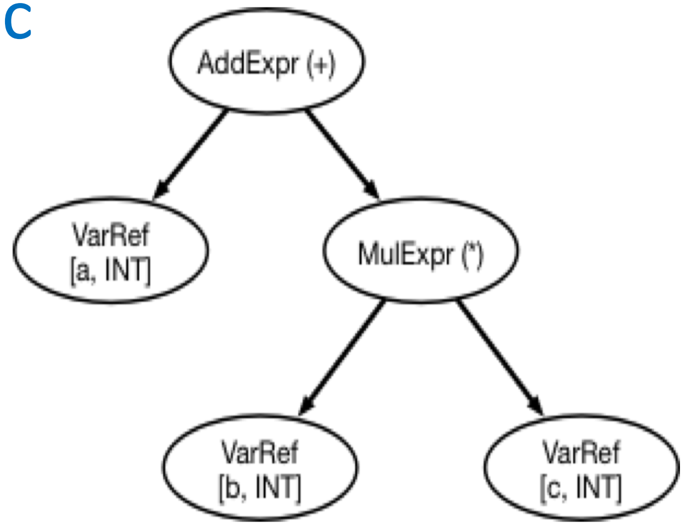
AST is Optional, but highly recommended!

Parse Tree vs Abstract Syntax Tree (AST)

Parse tree for a + b * c



AST for a + b * c




```
expr             -> expr_prefix factor
expr_prefix      -> expr_prefix factor addop | empty
factor           -> factor_prefix postfix_expr
factor_prefix    -> factor_prefix postfix_expr mulop | empty
postfix_expr     -> primary | call_expr
call_expr        -> id ( expr_list )
expr_list        -> expr expr_list_tail | empty
expr_list_tail   -> , expr expr_list_tail | empty
primary          -> ( expr ) | id | INTLITERAL | FLOATLITERAL
addop            -> + | -
mulop            -> * | /
```

Semantic Actions for generating AST

- Information in the AST is associated with various nodes in the parse tree.
- You can use semantic actions, just as you did in Step 3, to pass information "up" the parse tree to build up the AST.
- Instead of passing information about a declaration, you can instead pass partially constructed abstract syntax tree nodes (you may want to define a class or structure called *ASTNode*)

More details is in **D2L: Project → Step4 Instructions**

Use a sound design, describe it well in your report

IR: 3 Address Code (3AC)

- 3AC is an intermediate representation where each instruction has at most two source operands and one destination operand.
- Unlike assembly code, 3AC does not have any notion of registers. Instead, the key to 3AC is to generate *temporaries* -- variables that are used to hold the intermediate results of computations.
- For example, the 3AC for d := a + b * c will be:

```
MULTI b c $T1
ADDI a $T1 $T2
STOREI $T2 d
```

IR Node:

Opcode	First operand	Second operand	Result

IR: 3 Address Code (3AC)

ADDI OP1 OP2 RESULT (Integer add; RESULT = OP1 + OP2)

SUBI OP1 OP2 RESULT (Integer sub; RESULT = OP1 - OP2)

MULTI OP1 OP2 RESULT (Integer mul; RESULT = OP1 * OP2)

DIVI OP1 OP2 RESULT (Integer div; RESULT = OP1 / OP2)

ADDF OP1 OP2 RESULT (Floating point add; RESULT = OP1 + OP2)

SUBF OP1 OP2 RESULT (Floating point sub; RESULT = OP1 - OP2)

MULTF OP1 OP2 RESULT (Floating point mul; RESULT = OP1 * OP2)

DIVF OP1 OP2 RESULT (Floating point div; RESULT = OP1 / OP2)

STOREI OP1 RESULT (Integer store; store OP1 in RESULT)

STOREF OP1 RESULT (Floating point store; store OP1 in RESULT)

READI RESULT (Read integer from console; store in RESULT)

READF RESULT (Read float from console; store in RESULT)

WRITEI OP1 (Write integer OP1 to console)

WRITEF OP1 (Write float OP1 to console)

WRITES OP1 (Write string OP1 to console)

Opcode	First operand	Second operand	Result

Generating 3AC

- You can perform a post-order walk of the AST, passing up increasingly longer sequences of IR code called *CodeObjects*. Each code object retains three pieces of information:
  1. CODE: A sequence of *IR Nodes* (a structure representing a single 3AC instruction) that holds the code for this part of the AST (i.e., that implements this part of the expression)
  2. TEMP: An indication of where the "result" of the IR code is being stored (think: the name of the temporary or variable where the result of the expression is stored)
  3. TYPE: An indication of the type of the result (INT or FLOAT)

Generating Assembly

- Once you have your IR, your final task is to generate assembly code using an assembly instruction set called Tiny (see the *tinyDoc.txt)*
- Mapping is fairly straightforward: iterate over the list of 3AC you generated in the previous step and convert each individual instruction into the necessary Tiny code
  - Note: Tiny instructions reuse one of the source operands as the destination, so you may need to generate multiple Tiny instructions for each 3AC instruction).
- You will be using a version of Tiny that supports 1000 registers, so you can more or less directly translate each temporary you generate into a register (i.e. you don't have to worry about efficient register allocation).

What you need to do (Part 1)

- In this part, you will be generating assembly code for assignment statements, expressions, and READ and WRITE commands.
- Use the steps outlined above to generate Tiny code. Your code should output a list of tiny code that we will then run through the Tiny simulator to make sure you generated the right result.
- For debugging purposes, it may also be helpful to emit your list of IR code. You can precede a statement with a ; to turn it into a comment that our simulator will not interpret.

Do not need to add anything new to the grammar (.g4) file from Step 2

Notes

- All the inputs we will give you in this step will be valid programs.
- We will also ensure that all expressions are type safe: a given expression will operate on either INTs or FLOATs, but not a mix, and all assignment statements will assign INT results to variables that are declared as INTs (and respectively for FLOATs).
- In this step, we will only grade your compiler on the correctness of the generated Tiny code.
  - We will run your generated code through the Tiny simulator and check to make sure that you produce the same result as our code.
  - When we say result, we mean the <u>outputs of any WRITE statements in the program</u>.
- We will not check to see if you generate <u>exactly</u> the same Tiny code that we. In other words, we only care if your generated Tiny code <u>*works correctly*</u>.
  - You may generate slightly different Tiny code than we did.

Project Step 4 — Part 2: Control Structures

- This step builds on Part 1.
- Generate code for control structures (IF statements and WHILE loops)
- Note: as in part 1, we will only have one function in our program, main.
- You can assume that all variables are defined globally. There will not be any additional variables defined in main().

ASTs for Control Structures

- ASTs for control structures are, intuitively, simple: each control structure will have several children (3 in the case of an IF statement, etc.) that are themselves ASTs
- You already have working code for building an AST for statement lists
- All you have to do is create semantic actions for the control structures that "stitch together" the existing ASTs.

ASTs for Control Structures

- E.g: when you are generating code for an IF AST node, you know that the 3AC for the three children already exists.
  - All that is left is to put them together in the correct order and insert any necessary labels and jumps.
- The 3AC you will generate for labels looks like: `LABEL STRING`
- Unconditional jumps are easy: `JUMP STRING`
- Conditional jumps are a little bit tricky in our 3AC (and in Tiny): you need to generate the right kind of jump:
  - `E.g.` Greater than → `GT  OP1 OP2 LABEL`

More details is in **D2L: Project → Step4 Instructions**

What you need to do (Part 2)

- In this part, you will be generating assembly code for IF statements and WHILE loops.
- Use the steps outlined above to generate Tiny code. Your code should output a list of tiny code that we will then run through the Tiny simulator to make sure you generated the right result.
- For debugging purposes, it may also be helpful to emit your list of IR code. You can precede a statement with a ; to turn it into a comment that our simulator will not interpret.

Do not need to add anything new to the grammar (.g4) file from Step 2