Tutorial 2: An Expression Language Compiler

ECE 466/566 Spring 2025

Jan 27, 2025
Due: Feb 9, 2025
Due within two weeks of demonstration in class.
You are encouraged to make comments directly on this document.

1. Objective

Implement a compiler that will convert simple arithmetic expressions on registers and immediate values into assembly code. You may convert to an assembly language of your choice, but the instructions use LC-3.

2. Description

The Expression Language

The Expression Language makes it possible to write out simple expressions using Registers (R0 to R7) and immediate values using a convenient algebraic notation rather than assembly code. The language looks something like this:

```
R1 = (R0 + 5) - 3;
```

This would generate assembly code similar to this:

```
ADD R8, R0, 5    ; add R0 and 5
ADD R9, R8, -3   ; subtract 3
ADD R1, R9, 0    ; set R1 equal to R9
```

Note, we created R8 and R9 to hold temporary results. We will assume there are an infinite number of registers available starting at R8.

It also supports loads. If an expression is in brackets "[]", it means to treat the expression as an address and load from memory. For example:

```
R0 = [(R0+5)-3] + 10;
```

becomes something like this:

```
ADD R8,  R0,  5  ; add R0 and 5
ADD R9,  R8, -3  ; subtract 3
LDR R10, R9,  0  ; load from MEM[R9]
ADD R11, R10, 10 ; add 10
ADD R0,  R11, 0  ; put R11 into R0
```

Here's the last example.

```
R3 = R4 - (R1+R0);
```

becomes equivalent to the following LC-3 code:

```
   ADD R8, R1, R0    # R8 is a temporary for R1+R0
   NOT R9, R8        # R9 is a temporary for ~R8
   ADD R10, R9, 1    # R10 is a temporary for -(R1+R10)
   ADD R11, R4, R10  # R11 is the final result
   ADD R3, R11, #0   # Copy result to R3
```

Ls

Note, there is starter code for you in ***`https://github.com/jamesmtuck/ncstate_ece566_spring2025/tutorials/2`****. So you do not need to manually type all of this code if you don't want to. You may need to run* `git pull` *to get my latest changes. If my repo is a remote branch called ece566, use* `git pull --rebase ece566 main`.

If you're using CLion you can open this project by pointing CLion to the CMakeList.txt file in the tutorials/2 directory. It should be able to compile it for you. If you installed software on your own system, you may also need to install bison and flex if you have not already done so.

Here's a video to help set up CLion for the Tutorial so that you're ready to start coding (from 2022): Setup CLion Video.

3. The Scanner

The Tokens of the Language

Lexical Token	Pattern	Action
`REG`	`[Rr][0-7]`	`printf("REG: %d", atoi(yytext+1));`

`IMMEDIATE`	`[0-9]+`	`// should fit into an int printf("IMM:%d", atoi(yytext));`
`ASSIGN`	`=`	`printf("ASSIGN");`
`SEMI`	`;`	`printf("SEMI");`
`LPAREN`	`(`	`printf("LPAREN");`
`RPAREN`	`)`	`printf("RPAREN");`
`LBRACKET`	`[`	`printf("LBRACKET");`
`RBRACKET`	`]`	`printf("RBRACKET");`
`MINUS`	`-`	`printf("MINUS");`
`PLUS`	`+`	`printf("PLUS");`
`COMMENT`	`"//".*\n`	`printf("COMMENT");`
`ILLEGAL`	`.`	`yyerror("Syntax error!");`

Create a scanner that will recognize all of these tokens.

1. Create a new Flex input file called expr.lex.
2. Add a header to the file that will include stdio.h and math.h:

```
%{
     #include <stdio.h>
     #include <math.h>
%}
```

3. Create a region where you define the regular expressions (tokens):

```
%option noyywrap
%option debug
%% // begin tokens

"+"  { printf("PLUS"); }  // Use double quotes, not single

%% // end tokens
```

4. Use a single `printf` as your actions right now so that you can see what's happening.

5. Add one or two regular expressions and test it. Remember to put double-quotes around special characters (like PLUS and MINUS) since they are used to describe regular expressions. If your working from scratch, you can test that your code compiles as follows:
```
flex -oexpr.lex.c expr.lex
gcc -o expr expr.lex.c -ll
```

-ll may produce a warning. However, you may not need -ll. This says to link with the flex library. This is only necessary if you choose not to implement your own main and it will grab main from the library. But, we can just stick our own main at the bottom of the file and it will work just fine, which we did. So you can drop -ll.

Or, you can install it if it is missing. I realized it's missing from Docker so you may have to run this command from you docker container or add it to the Dockerfile and re-build the container:

apt-get install libfl-dev

```
int main(int argc, char *argv[])
{
  // all the rules above are combined into a single function
  // called yylex, we call it to trigger
  // the scanner to read the input and match tokens:

  yylex();
  // yylex has a return value, but we ignore it for now.

  return 0;
}
```

If you start from my repo, you can just enter:
```
make
```

6. Add more regular expressions. **Test frequently.**
7. Once you support all of the tokens, add some additional features:
    a. Ignore whitespace. For example: `[\t\n ]`
    b. Detect characters that shouldn't be there and report an error. Add a rule to match any other character and report an error.
8. When a token matches, there's a global pointer to the string that matched, called yytext that we can use to print out the matching string. For example:
```
[Rr][0-7]  { printf("REG: %s ", yytext); }
```

Change some of your rules to print out the value that was scanned along with the token name.

Questions:

A. What happens if you input something like this:? `R0 R23 453 -16 -17` What tokens do you get?
When the input R0 R23 453 -16 -17 is written to expr.lex file, in order the tokens produced are "REG: 0, Syntax error! , REG: 2, IMM: 3, Syntax error! , IMM: 453, Syntax error!, MINUS, IMM: 16, Syntax error!, MINUS, IMM: 17. (there are several "Syntax error!" statements due to the space between each input).

B. What happens if you don't explicitly detect errors on other unexpected characters (like $ or *)?
If you do not explicitly detect errors on other unexpected characters, the exper.lex file will read this as a syntax error. Since characters like "$" and "*" are not defined in the previous set of tokens, it will be read as a syntax error.

C. Does this scanner guarantee that immediates are within a particular range?
No, this scanner does not guarantee that immediates are within a particular range. The immediate tokens are defined within [0-9]+, so it can vary from an immediate value of 13 to an immediate value such as 98997.

D. Could you modify the scanner with additional functionality, like support for scanning other arithmetic operators?

Yes, you can modify the scanner with additional functionality for scanning other arithmetic operators. A token definition can be added for division ("/"), multiplication ("*"), and even modulo ("%").

4. The Parser

The Grammar

The grammar shown below supports the kind of statements we want:

```
program:   REG ASSIGN expr SEMI
;

expr:        IMMEDIATE
           | REG
           | expr PLUS expr
```

```
            | expr MINUS expr
            | LPAREN expr RPAREN
            | MINUS expr
            | LBRACKET expr RBRACKET
;
```

The first rule, `program`, says that we must match an assignment statement to have a legal program. The second rule describes all of the ways we can form an expression.

Create a parser that will recognize the grammar.

1. Create a new Bison input file called expr.y.
2. Add a header to the file that will include stdio.h and math.h:
   ```
   %{
        #include <stdio.h>
   %}
   ```
3. Create a region where you define the grammar:
   ```
   %% // begin grammar



   %% // end grammar
   ```
4. Before the grammar section begins, name all of the tokens:
   ```
   %token REG ASSIGN MINUS PLUS ...
   ```

   We can assign these names however we want. They will become #defines in a header file created by bison that must be included from expr.lex. The inclusion of the header file is what allows the scanner and parser to communicate with each other.

5. Specify equal precedence and left groupings for addition and subtraction:
   ```
   %left  PLUS MINUS
   ```
   This goes in the region before the grammar (before first %%) after the %token directive. This prevents our grammar from being ambiguous.

6. Add the grammar to the file. In each action, **add a printf that prints the rule that was matched**. This will help us observe what's happening for now, but later we'll remove it. For example, something like this:

```
program:   REG ASSIGN expr SEMI
                           { printf("REG ASSIGN expr SEMI\n"); }
;
```

7. Modify the Flex file, expr.lex:
    - Make sure it includes expr.y.hpp. The expr.y.hpp file will be generated automatically by bison later, so don't worry about where it comes from. Also, since it's part of the code that gets copied to the final scanner code, flex will just ignore this for now.
    - Second, in each action, force each rule to return the token that was found, something like this: `{ return REG; }`
      Important: *this name must match the name in the token command placed in the bison file.*
    - The name of the token should match the one declared in the token line above.
8. Now, you're ready to test the whole parser. If you're developing from scratch, do this:
    - `bison -y -d -o expr.y.c expr.y`
    - `flex -oexpr.lex.c expr.lex`
    - `gcc -o expr expr.y.c expr.lex.c -ly -ll`

   Otherwise, for my repo code:

   ```
   make
   ```

9. Test the parser by running the expr program and inputting both legal and illegal programs. If you made the rules with the prints as I described, then here are some test inputs and outputs for you to try:

Input	Output
R1;	syntax error
R0 = 0;	IMMEDIATE REG ASSIGN expr SEMI
R1 = R1 + 1;	REG IMMEDIATE expr PLUS expr REG ASSIGN expr SEMI
r1 = [r5+ 3] - 2+0;	REG IMMEDIATE expr PLUS expr LBRACKET expr RBRACKET IMMEDIATE expr MINUS expr IMMEDIATE expr PLUS expr REG ASSIGN expr SEMI

Does the order of the printfs appear to be in a pattern? Does it follow a logical order?

10. Make up some inputs of your own with the goal of making some parse and some fail to parse.

Questions

E. Create your own input expression different from the ones above and write down what you think the output will be. Then compare to the actual output. Were you right or wrong? Describe what you observed or learned from this.
Example Input Expression: r3 = (r2 + 2) + [r1 + 1]
What I think the output will be:
REG, IMMEDIATE, expr PLUS expr, LPAREN expr RPAREN, REG, expr PLUS expr, LBRACKET expr RBRACKET.
Actual Output: REG, IMMEDIATE, expr PLUS expr, LPAREN expr RPAREN, REG, expr PLUS expr, LBRACKET expr RBRACKET. - THIS IS CORRECT

What I observed from this exercise was that the parenthesis/bracket statements are called after the values/registers that are held within them. I originally thought that the registers within a bracket would be called after, however it makes sense why the brackets are called following the content.

Do you see a pattern in the order of the printfs shown above? What is it?

There is a pattern in the order of the printfs shown above. The printfs begin with the type of token and then follow up with the operation associated with the tokens (seems to follow some sort of order of operations). This pattern remains consistent with the inputs given in this tutorial and remains consistent with other inputs.

F. [ECE566] Show how you would extend the grammar to support a store instruction.

5. Passing Information from the Scanner to the Parser

In our preliminary scanner, we were able to see the value associated with a token. For example, we knew that in this token:

R3

We were given register **3**. We computed the **3** using `atoi(yytext+1)`. However, that is now commented out and we are no longer using that information.

```
[Rr][0-7]  { /*printf("REG: %d ", atoi(yytext+1));*/ return REG; }
```

However, if we are going to generate assembly code, we must find a way to pass this information along to the parser, otherwise we don't know which register is being referenced. The REG token is not enough to figure out what is desired.

One idea is to change the return type, and do something like this:

```
[Rr][0-7]  { return atoi(yytext+1); } // don't do this!
```

However, this doesn't work. We must return the token name as defined by the parser, not some arbitrary value. That's just the way Bison and Flex work together. (If we wrote our own parser from scratch we wouldn't face this particular limitation, but we also would have a ton more code to write before even worrying about this!)

Another option is to put the value in a global variable. Fortunately, Bison makes it easy to do just that. We can tell Bison that we want to associate a variable/value with certain pieces of the grammar so that the parser knows the full context of the token. In this case, in addition to knowing that we have a register token (REG), we also want to know what the register number is.

Union and Type Directive

In the Bison file, we let the parser know that we want to associate some variables with tokens using a combination of a union declaration and a %type directive. The union just declares a union type in C that we can reference from code. The %type directive associates a field in the union with the token or non-terminal of interest.

For example, we wish to associate every REG with an integer that holds the register number. We do that as follows:

```
%union {
  int reg;
}
// Put this after %union and %token directives
%type <reg> REG
```

Now, every REG is associated with an integer that will hold its value. To take advantage of this information, we modify both the Flex rule and the Bison rule associated with REG. In the Flex file:

```
[Rr][0-7]  { yylval.reg = atoi(yytext+1); return REG; }
```

Here `yylval` is a variable that's declared from the union we defined earlier. Hence, `yylval` has a field name `reg`. We can set that field to have any integer value we want. In this case, we want it to be the number of the register.

On the other side in the parser, things are not quite as obvious. In the Bison file, we reference the reg field using a positional specifier, `$1`:

```
expr:  REG  { printf("REG (%d)\n",$1); }
```

The $1 comes from the fact that REG is the first token of the rule. We are saying, basically "get the value associated with the first token." Since we've already specified that any REG token is associated with the reg field, Bison knows that we must want yylval.reg associated with the REG.

We could do this same procedure for every token in the grammar that has some value we want to associate with it. IMMEDIATE, in particular, should be treated in a similar way. Go back and make similar changes to support IMMEDIATE.

```
expr:  IMMEDIATE  { printf("IMMEDIATE (%d)\n",$1); }
```

Don't forget to update the rule in your scanner. Now, check your outputs again.

Input	Output
R1;	syntax error
R0 = 0;	IMMEDIATE (0) REG (0) ASSIGN expr SEMI
R1 = R1 + 1;	REG (1) IMMEDIATE (1) expr PLUS expr REG (1) ASSIGN expr SEMI
r1 = [r5+ 3] - 2+0;	REG (5) IMMEDIATE (3) expr PLUS expr LBRACKET expr RBRACKET IMMEDIATE (2) expr MINUS expr IMMEDIATE (0) expr PLUS expr REG (1) ASSIGN expr SEMI

6. Syntax Directed Translation (video)

During parsing, we want to execute actions that serve to interpret the language and generate the appropriate output. There are a few key things we need to determine:

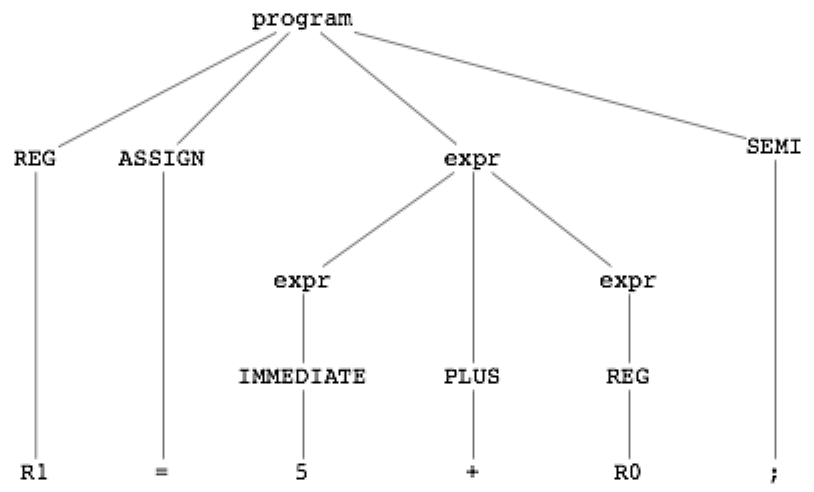
1. What action should be taken on each rule?
2. What data value or structure is needed to represent the terminals and non-terminals in the grammar?

Thinking Through the Actions

Think about how each rule in the grammar helps partially produce the desired output. For example, consider this example program:

```
R1 = 5 + R0;
```

The parse tree will look like this:



It might produce the following output:

```
# Output of expr: IMMEDIATE
AND R8, R8, 0
ADD R8, R8, 5
# Output of expr: expr PLUS expr
ADD R9, R0, R8
# Output of REG ASSIGN expr SEMI
ADD R1, R9, 0
```

The first rule that can match is `expr:IMMEDIATE`. (As a general rule, actions for the children nodes must execute before parent nodes in the parse tree. So, the rules for IMMEDIATE(5), PLUS, and REG(R0) must execute first.)

Let's consider an implementation of `expr: IMMEDIATE`.

```
expr:  IMMEDIATE   {
   regCnt++; // get the next free register
   printf("# Output of expr: IMMEDIATE\n");
   printf("AND R%d, R%d, 0",regCnt,regCnt);      // clear a register
   printf("ADD R%d, R%d, %d",regCnt, regCnt, $1); // add immediate
   $$ = regCnt; //specify which register holds the result
}
```

There are two important things in this code. First, we refer to **$1** to get the immediate, which will be passed in from the scanner. Hence, the IMMEDIATE token should be an integer value. Second, we set the result (**$$**) to be the **register number** holding the immediate value. This is a natural thing to do (although somewhat inefficient). It also leads to a useful strategy: **treat each occurrence of `expr` as a register**. We can just use an integer to hold the register number.

The next line of assembly would be handled by these two rules:

```
expr:   REG
expr:   expr PLUS expr
```

It happens in two parts. First, the register R0 is matched as an expr. The scanner will send over the register number, and using what we learned in the previous rule, we just make it the result of the rule, like so:

```
expr:   REG   { $$ = $1; }
```

Then, we handle the add rule. We would probably want it to work something like this:

```
expr:   expr PLUS expr  {
    printf("# Output of expr: expr PLUS expr\n");
    regCnt++; // get the next free register
    printf("ADD R%d, R%d, R%d",regCnt, $1, $3);
    $$ = regCnt; // the result of this rule is the register
                 // holding the result
}
```

In this code, we continue treating `expr` as a register by referring to its register number. We generate an ADD instruction that adds the two registers into a third register.

`$1` and `$3` are registers because they are an `expr`. No matter what rule an expr occurs in, it will always be a register. The third register is created just by incrementing the global variable, regCnt.

We can continue this strategy to implement the rest of the rules. Here's a recap:

1. `expr` rules are associated with an integer that refers to a register number.
2. Each rule produces as its result a register number and assigns it to `$$`.
3. When expr appears on the right side (inside the rule) then `$n` refers to a register number.

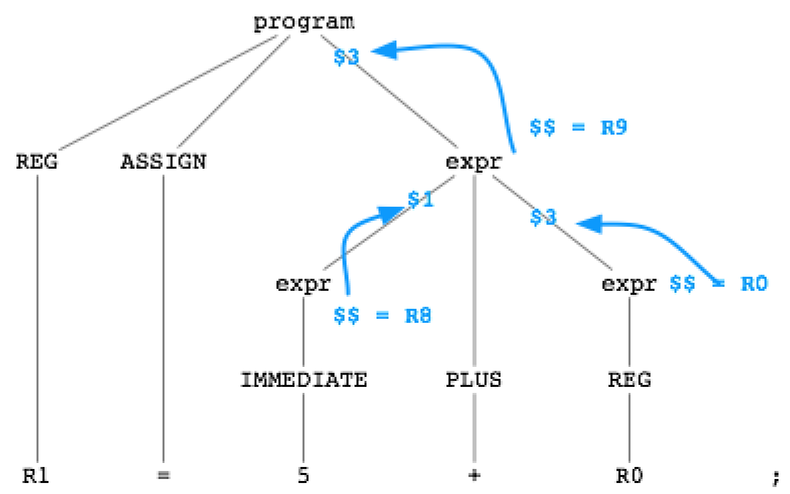
Look through the other rules and continue the assumption that all `expr` rules produce a register number as its result. If it works for all the rules, then we can write all the actions in a similar way.

A couple of other things to remember. We changed the scanner so that REG and IMMEDIATE are passed to the parser with integers. REG comes from the scanner with a register number. IMMEDIATE should come with its associated integer value.

How Results Are Passed from One Rule to the Next

A common challenge when looking at parsers for the first time is understanding how data is passed from one rule to the next. We know we can set the output of a rule as `$$`. And, we can get the value associated with each part of a rule as `$n`. But, how does it all fit together?

The way I think about it is in terms of the parse tree. Each rule produces an output and passes it up the parse tree. *Keep in mind that children nodes in the parse tree must complete their actions before parent nodes.* This is guaranteed to always be true! So, when we write the action for a rule, we know that each sub-rule has already executed and provided its result.



In the picture above, I'm showing three cases. The `IMMEDIATE` rule passes the register number 8 up as its result. But, in the `expr PLUS expr` rule, R8 is seen as $1 since it has already been evaluated and is associated with the left-hand-side of the addition.

The `REG` rule passes register number 0. In the `expr PLUS expr` rule, it's obtained as $3 since it's the right-hand-side of the addition.

Implement All the Actions

1. Decide on an appropriate type for each token that represents something that needs to appear in the output (i.e. registers and immediates).
2. Declare the union and associated fields, for example:
   ```
   % union {
        int regno;
        int imm;
        // anything else
   }
   %token <regno> REG
   %token <imm> IMMEDIATE
   %type <regno> expr
   ```
3. Remember to declare and initialize global variables or functions you need at the top of the Bison input file.
4. Implement the actions associated with each rule of the grammar. Test as you go to ensure it's working as expected. Make sure to create tests that only use the rules you've written actions for.

Questions

H. Is the assembly you generated efficient? Give an example of your output to make your case. How do you think this relates more broadly to code generation and optimization techniques?

The assembly that was generated is somewhat efficient, however it can be more efficient. For example, repeating the same register (example output from my script: “AND R1, R1, 0…ADD R1, R1, 5). The regCnt counter increments whenever a new register is created. This can potentially lead to excessive registers produced from the compiler code in this tutorial. This is common when the same value is repeated multiple times. This relates more broadly to code generation and optimization techniques because the repeated registers relates to scheduling instructions in a sequence, or register allocation.

l. Instead of printing the assembly while parsing, what advantages could be had by building a data structure to represent the instructions?
Instead of printing the assembly while parsing, the advantages that occur when building a data structure to represent the instructions include reusing register values, flexibility in storing certain instructions for future assembly language inputs, and even making debugging simpler.
    a. [ECE566] Describe at least one thing that would have to change in your code to make this possible.

Grading

Submit answers to the questions and your final code to Tutorial 2 on Moodle/GradeScope.

[50 points] Answers to questions checked for completeness.
[50 points] Code checked for completeness.

Appendix

Other resources:

- Bison Manual
- Flex Manual