UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO.

Team Gremlins

Compiler

ING. Norberto Jesús Ortigoza Márquez

Barrientos Veana Luis Mauricio.

González Pacheco Leonardo Alonso.

Martínez Matías Joan Eduardo.

Rosales Romero Ricardo.

Project Scope C Compiler, Part 2

Failures detected for the first delivery:

- The organization was not efficient and the goals to be achieved had to be rethought due to poor organization.
- The use of used tools was not 100% mastered and therefore there were problems in understanding the way of working.
- The implementation of all the parts of the compiler and understanding its operation caused confusion at first but the errors were debugged.

Settings for the second release

- In the way of organization there was more communication with the members of the team.
- The tools for making the compiler were better mastered.
- More realistic dates and objectives were set.
- There were changes but errors could be fixed.

Activity planning

Tasks	Week 1	Week 2	Week 3	Week 4
Receive feedback from customer and subject matter expert	🟧			
Read documentation from Nora Sandler.		🟩		
Implement new lexing operators in the token list.			🟦	
Add to tree AST located in Parsing expressions.				🟥
Debug the code and upload changes to GIT and GIT HUB.				🟥

Subject

In the first part we learned different things about integers in the compiler. Our second goal was to add three unary operators (operators that only take one value).

For example:

Negation (-)

-5=0-5. In other words, it’s a regular negative number.

Bitwise complement (~)

This flips every bit in a number2.For example:

4 is written as 100 in binary.

The bitwise complement of 100 is 011 so ~4=3.

Logical negation(!)

The Boolean “not” operator. This treats 0 as “false” and everything else as “true”.

!0 = 1

! (anything else) = 0

In the last installment we created a compiler with three stages: a lexer, a parser, and a code generator. Now in this release we updated each stage to handle these new operators.

Lexing

We just needed to add each of these operators to our list of tokens. Here’s the list of tokens: tokens from last release are at the top, and new tokens are bolded at the bottom.

- Open brace {
- Close brace}
- Open parenthesis \(
- Close parenthesis \)
- Semicolon;
- Int keyword int
- Return keyword return
- Identifier [a-zA-Z]\w*
- Integer literal [0-9] +
- **Negation -**
- **Bitwise complement ~**
- **Logical negation!**

We can process these new tokens exactly the same way as the other single-character tokens, like braces and parentheses.

<u>Parsing</u>

Last release we defined several AST nodes, including expressions. We only defined one type of expression: constants. This step, we add another type of expression, unary operations. The latest set of definitions is below. Only the definition of exp has changed.

Now, an expression can take one of two forms - it can be either a constant, or a unary operation. A unary operation consists of the operator (e.g. ~), and the operand, which is itself an expression.

Our definition of expressions is recursive - expressions can contain other expressions!

This is an expression: !3

So is this: !~-4

So is this: !!!!!!!-~~~-!3

<u>Code Generation</u>

Negation and bitwise complement are super easy; each of them can be accomplished with a single assembly instruction.

neg negates the value of its operand. Here’s an example:

```
movl    $3, %eax    ;EAX register contains 3
neg     %eax        ;now EAX register contains -3
```

Of course, we need to calculate a value before we can negate it, so we need to recursively generate code for the inner expression, then emit the neg instruction.

not replaces a value with its bitwise complement. We can use it exactly the same way as neg.

Logical negation is a little more complicated. Return! exp is equivalent to:

```
if (exp == 0) {
    return 1;
} else {
    return 0;
}
```

Unlike the other operations, which were straightforward bit manipulation, this requires some conditional logic. We can implement it using cmpl, which compares two values, and sete (“set if equal”). sete sets its operand to 1 if the result of the last comparison was equal, and 0 otherwise.

Comparision and conditonal instructions like cmpl and sete are a little weird; cmpl doesn’t explicitly store the result of the comparison, and sete doesn’t explicitly refer to that result, or to the values being compared. Both of these instructions - and all comparison and conditional instructions - implicitly refer to the FLAGS register.

As the name suggests, the contents of this register are interpreted as an array of one-bit flags, rather than a single integer. These flags are automatically set after every arithmetic operation. The only flag we care about right now is the zero flag (ZF), which is set on if the result of an operation is 0 and set off otherwise.

cmpl a, b computes (b - a) and sets FLAGS accordingly. If two values are equal, their difference is 0, so ZF will be set on if and only if the operands to cmpl are equal. The sete instruction uses ZF to test for equality; it sets its operand to 1 if ZF is on, and 0 if ZF is off. In fact, setz (“set if zero”) is another mnemonic for the same instruction.

The last gotcha here is that sete can only set a *byte*, not an entire word. We’ll have it set the AL register, which is just the least-significant byte of EAX. We just need to zero out EAX first; since the result of ! is always 0 or 1, we dont want to leave any stray higher bits set.

So that was a long explanation, but you can actually implement! in just three lines of assembly:

```
<CODE FOR exp GOES HERE>
cmpl   $0, %eax    ;set ZF on if exp == 0, set it off otherwise
movl   $0, %eax    ;zero out EAX (doesn't change FLAGS)
sete   %al         ;set AL register (the lower byte of EAX) to 1 iff ZF is on
```

To convince ourselves that this is correct, let’s work through this with the expression !5:

1. First, we move 5 into the EAX register.
2. We compare 0 to 5. 5!= 0, so the ZF flag is set to 0.
3. The EAX register is zeroed out, so we can set it in the next step.
4. We conditionaly set AL; because ZF is 0, we set AL to 0. AL refers to the lower bytes of EAX; the upper bytes were also zeroed in step 3, so now EAX contains 0.

Conclusions

We will have to understand the concepts of negation and complement at the bit level; The way to implement these elements in the code was to follow the instructions of Nora Sandler where we explained step by step how to do it.

Unary operators and specific expressions are added in the AST tree earlier in the analysis part.

Little by little, the way in which elixir facilitates these tasks is understood and as a team we are going to solve the problems that are shown.

References

https://norasandler.com/2017/12/05/Write-a-Compiler-2.html