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.

Index

Introduction

Client Norberto Jesus Ortigoza Marquez requires a Commands that compile C language codes. The codes that need to be compiled. Provided by the same client with the purpose of all of them being compiled correctly generating the corresponding executables.

Norberto also requests that the program be created under other specific which are detailed in the requirements. Also ask for an installation manual and use so that the program can be used by any user with basic command-line knowledge. Finally, the client offers a personal repository on GitHub in order to give monitoring the project and its generated versions. Also, making a calendar of deliveries of project progress.

When making deliveries, Norberto will request explain the architecture of the program and the functionality of the source code also show a suite of tests attached to the codes provided by the client

Failures detected for the forth 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 ultimate 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.

- It was checked that the last modifications were correct for the final delivery of the compiler project

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.				
Planning compiler improvements				
Debug the code and upload changes to GIT and GIT HUB.				

Latest code modifications

We're adding some boolean operators (`&&`, `||`) and a whole bunch of relational operators (`<`, `==`, etc.). The three tests whose names start with `skip_on_failure_` use local variables, which we haven't implemented yet. When you run the test suite, they should show up as `NOT_IMPLEMENTED` rather than `FAIL` in the results, and they shouldn't count toward the total number of failures. Once you've implemented local variables, these tests should pass.

We're adding eight new operators this week:

- Logical AND `&&`
- Logical OR `||`
- Equal to `==`
- Not equal to `!=`
- Less than `<`
- Less than or equal to `<=`
- Greater than `>`
- Greater than or equal to `>=`

As usual, we'll update our lexing, parsing, and code generation passes to support these operations.

Lexing

A grammar is the set of rules that define the words that a language recognizes, accepts and / or generates from a given alphabet. It is the first phase of the compiler and is known by the name of Scanner. It transforms a set of characters that are read from the source program one by one as input and output lexical components or also known as

tokens, which will then be used by the parser. And it is responsible for recognizing identifiers, keywords, constants, operators, etc.

The lexical parser works at the request of the parser by giving it a lexical component every time the parser needs it. The lexical components are specified using regular expressions.

Each new operator corresponds to a new token. Here's the full list of tokens we need to support, with old tokens at the top and new tokens in bold 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]+`
- Minus `-`
- Bitwise complement `~`
- Logical negation `!`
- Addition `+`
- Multiplication `*`
- Division `/`
- **AND** `&&`
- **OR** `||`
- **Equal** `==`
- **Not Equal** `!=`
- **Less than** `<`
- **Less than or equal** `<=`
- **Greater than** `>`
- **Greater than or equal** `>=`

We have a lot more precedence levels, which means our grammar will grow a lot. However, our parsing strategy hasn't changed at all; we'll handle our new production rules exactly the same way as the old rules for `exp` and `term`.

Parsing

The second phase of the compiler is known as parsing or parsing. The parser uses the tokens produced by the lexical parser to create a tree structure that represents their grammatical structure.

This analysis is necessary to determine if a series of tokens supplied by the lexical analysis are valid in a certain language, that is, if the sentence has the correct structure or form. But not all syntactically correct sentences are valid, semantic analysis will be needed to determine this accurately.

A parser takes the output of the lexical parser in the form of a string of tokens and parses it using the production rules to detect errors in the code. The output of this is a syntax tree. In this way, parsing performs two tasks, grammatically analyzing the code, finding errors, and generating a syntax tree as output.

The parser must analyze all the code even if there are errors, this is why error recovery strategies are used

Here are our all binary operators, from highest to lowest precedence[1]:

- Multiplication & division (`*`, `/`)
- Addition & subtraction (`+`, `-`)
- Relational less than/greater than/less than or equal/greater than or equal (`<`, `>`, `<=`, `>=`)
- Relational equal/not equal (`==`, `!=`)
- Logical AND (`&&`)
- Logical OR (`||`)
    - We'll add a production rule for each of the last four bullet points. The new grammar is below, with changed/added rules bolded.

- `<program> ::= <function>`
- `<function> ::= "int" <id> "(" ")" "{" <statement> "}"`
- `<statement> ::= "return" <exp> ";"`
- **`<exp> ::= <logical-and-exp> { "||" <logical-and-exp> }`**
- **`<logical-and-exp> ::= <equality-exp> { "&&" <equality-exp> }`**
- **`<equality-exp> ::= <relational-exp> { ("!=" | "==") <relational-exp> }`**
- **`<relational-exp> ::= <additive-exp> { ("<" | ">" | "<=" | ">=") <additive-exp> }`**
- **`<additive-exp> ::= <term> { ("+" | "-") <term> }`**
- `<term> ::= <factor> { ("*" | "/") <factor> }`
- `<factor> ::= "(" <exp> ")" | <unary_op> <factor> | <int>`

- `<unary_op> ::= "!" | "~" | "-"`

We had to rename it because `<exp>` now refers to logical OR expressions, which now have lowest precedence.

now you'll need `parse_relational_exp`, `parse_equality_exp`, etc. Other than handling different operators, these functions will all be identical.

And for the sake of completeness, here's our AST definition:

```
program = Program(function_declaration)

function_declaration = Function(string, statement) //string is the function name

statement = Return(exp)

exp = BinOp(binary_operator, exp, exp)

| UnOp(unary_operator, exp)

| Constant(int)
```

This is identical to last week, except we've added more possible values of `binary_operator`.

We can modify this slightly to implement `==`:

```
<CODE FOR e1 GOES HERE>

push    %eax            ; save value of e1 on the stack

<CODE FOR e2 GOES HERE>

pop     %ecx            ; pop e1 from the stack into ecx - e2 is already in eax

cmpl    %eax, %ecx      ;set ZF on if e1 == e2, 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
```

The `sete` instruction is just one of a whole slew of conditional set instructions. There's also `setne` (set if not equal), `setge` (set if greater than or equal), and so on. To implement `<`, `>`, and the other relational operators, we can generate exactly the same assembly as we used for `==` above, just replacing `sete` with the appropriate conditional set instruction. Easy!

we need the sign flag (SF), which is set if the result of an operation is negative, like so:

```
movl $0, %eax ;zero out EAX
movl $2, %ecx ;ECX = 2
cmpl $3, %ecx ;compute 2 - 3, set flags
setl %al      ;set AL if 2 < 3, i.e. if 2 - 3 is negative
```

Now let's talk about `&&` and `||`. I'll use `&` and `|` to indicate bitwise AND and OR, respectively.

Code Generation

The code generator is the phase through which a syntactically correct program is converted into a series of instructions to be interpreted by a machine. The input for this phase is represented by a syntax tree, the memory locations are selected for each of the variables used by the program. Then each of the intermediate instructions is translated into a sequence of machine instructions that executes the same task. In such a way that it generates assembly code.

Learnings

This project left us too many learnings, the one we value the most was collaborative work and teamwork, they are two different ways of working, but together they help the project to be carried out correctly.

Also we learned too much how to think to help a user make a code and our compiler can give him some help.

Finally we leave with a lot of knowledge and see from another point of view a code that compiles other codes and see what mistakes can be made during a code creation.

Conclusions

By understanding the latest compiler tests using local variables, it was possible to see how a short circuit is made

Also adding the new tokens, increased the list that we had previously in the lexer

Finally, we can say that it was not an easy job, it had its complications like any project, also because of the situation in which we find ourselves, the times and communication with our coworkers was more difficult, but we were able to carry out our project, now that a way out and the support of all the members was sought.

References

- https://norasandler.com/2017/12/28/Write-a-Compiler-4.html
- http://blog.pucp.edu.pe/blog/tito/2019/01/05/crea-tu-propio-compilador-casero-parte-1/
- http://blog.pucp.edu.pe/blog/tito/2019/01/13/crea-tu-propio-compilador-parte-4-creando-un-analizador-lexico/
- http://blog.pucp.edu.pe/blog/tito/2019/02/09/crea-tu-propio-compilador-parte-6-analisis-sintactico/