Conditional Statements and Loops

Lecture #07

Charles Averill

Practical Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

Last week we added conditional statements to our language:

```
(base) charles@nostromo:~/Desktop/ecco$ cat examples/test1
print 7 == 7;
print 7 != 9;
print 7 < 9;
print 7 <= 9;

print 9 > 7;
print 9 >= 7;

int x;
x = 7 >= 7;
print x;
x = 7 <= 7;
print x;

// This is the same as (1 > 1) < 2
print 1 > 1 < 2;
(base) charles@nostromo:~/Desktop/ecco$ ./scripts run examples/test1
 && clang test1.ll -o test1 && ./a.out
-----RUN-----
1
1
1
1
1
1
1
1
1
1
```

The Goal

Today, we'll be covering one of my favorite topics in this course: conditionals and loops.

Up to now, we've implemented some very fundamental features that most imperative languages should have, but we haven't implemented any control structures yet. This will be our first, so get excited! We will implement if statements, while loops, and for loops.

Quick BNF Update - If Statements

```
statements: "{" "}"
          | "{" statement "}"
          | "{" statement statements "}"
statement: "print" expression ";"
         | "int" IDENTIFIER ";"
         | IDENTIFIER "=" expression ";"
         | if_head statements
         | if_head statements "else" statements
         | while_head statements

if_head: "if" "(" expression ")"
while_head "while" "(" expression ")"

expression: add_expression
```

Quick BNF Update - If Statements

Notice that we wrap all our statements with braces. We do this to solve the "dangling else problem".

The dangling else problem arises in parsing and implementations of it can change the semantic meaning of a program. It occurs when nested "if"s and an "else" lead to an ambiguous attachment of the "else" to one of the "if"s. To illustrate with some arbitrary language:

```
if (x < y):
if (a != b):
    echo("Hello World!")
else:
    echo("Foo Bar!")
```

Which "if" does the "else" belong to? It's implementation-specific! Braces solve this problem by explicitly wrapping code blocks.

If Statements: The Plan

- Add tokens for "if", "else", parentheses, and braces
- Add an if-statement parser (and update our generic statement parser along the way)
- Implement generators for LLVM labels and jumps
- Add a testing suite to our language. We have a lot of features now, so we should have something more versatile than manual testing

Statement Parser Updates

We've had to make some changes to the statement parser to support conditional statements.

The issue is that we will have IF AST nodes that need to hold an expression (for the condition), and potentially two `statements` trees, one for the inner block of the "if", and one for the inner block of the "else".

First of all, we're going to have to add a `middle` ASTNode child in the ASTNode class.

Secondly, we're going to go back on our statement parser optimization that utilized Python's generator functions to save memory. Because we're chaining statement blocks together in ASTs, this isn't a feasible approach any more, at least not until we start implementing functions.

Statement Parser Updates

Thirdly, we're going to start "gluing" ASTs together. Because we have specific blocks of code that are context-sensitive, we need to glue statements together into one big AST than can be used as the child of more complex statements, like "if".

So, we'll say we have "root" and "left" AST Nodes while parsing statements. `root` is going to be the output of all of our individual statement parsers, like `assign_statement`.

Whenever we reach a right brace ("}"), we will return our `left`, if it exists. `left` gets built up over time, because it will glue itself to `root` if a right brace is not encountered. This process allows us to chain sub-ASTs together into one larger AST.

Finally, we write our special if-statement parser.

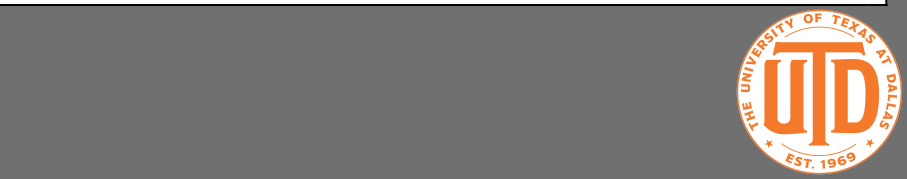
Code Generation - If Statements

LLVM's branching paradigm is fairly similar to most others that you've seen. There is only one branch instruction, `br`, that handles both conditional and uncoditional jumps:

```
; Unconditional jump to label L1
br label %L1
; Conditional jump
br i1 %Condition, label %TrueLabel, label %FalseLabel
```

One thing to note is that LLVM does **not** allow for fallthroughs. Therefore, whenever we generate a label we will often generate a jump statement to the label first, like so:

```
br label %L1
L1:
```



Converting if-else statements to LLVM

This pseudocode and LLVM are isomorphic to each other:

```
if condition:
    do_something_1
else:
    do_something_2
```

```
; Compute condition
br i1 %condition_register, label %TRUE_LABEL, label %FALSE_LABEL
TRUE_LABEL:
...          ; do_something_1
br label %END
FALSE_LABEL:
...          ; do_something_2
br label %END
END:
```

Code Generation - If Statements

Looking at `ast_to_llvm`, we've made a few structural changes.

First, I got tired of writing `[ASTNode].token.type` and added a `type` property.

Second, we added a `parent_operation` parameter to the function. Whenever we recursively call this function, we'll pass in our curren't node's TokenType. We do this because currently, we're enforcing that the condition in an IF statement uses a conditional operator, so when we generate code for those operators we need to know if we're generating code for a binary expression or for a conditional jump, since they behave differently.

Let's take a look at the code generator for if statements now.

Code Generation - If Statements

We've added an `LLVM_LABEL_INDEX` variable that keeps an eye on which label we can generate next. We're going to number them like the virtual registers for easy enumeration. We'll also use an "L" prefix for readability.

When generating if-statement code, we:

1. Generate code to compute the condition, followed by a conditional jump
2. Generate the "true" code block, followed by an unconditional jump to the "end" label, which will also be the "false" label if there is no else clause
3. Generate the "false" code block followed by an unconditional jump to the "end" label

Interjection

I've added a testing suite that uses `pytest` to run code for a bunch of different test programs and make sure that their output matches the desired output. This is a super simple setup and most popular languages have some similar unit testing framework

While Loops: The Plan

- Add "while" token
- Add a while-loop parser
- Generate looping LLVM

While Loop Updates

Our while loop parser is going to be almost identical to our if-statement parser, except that there is no "else" clause.

Similarly, our code generator is going to be almost identical to our if-statement generator, although the structure will be different:

```
br label %CONDITION_LABEL
CONDITION_LABEL:
; Compute condition
br i1 %condition_register, label %INNER_BLOCK, label %END
INNER_BLOCK:
...
br label %END
END:
```

While Loop Generator

1. Jump to our condition label
2. Generate our condition label
3. Generate our condition computation code followed by a conditional branch to the end label
4. Generate our inner block code followed by an unconditional branch to our condition label
5. Generate our end label

For Loops: The Plan

- Add "for" token
- Add a for-loop parser
- ...

That's it! For loops are a special case of while loops, so they're super easy to implement. For example, these two loops are identical:

```
int i;
for (i = 0; i < 10; i = i + 1) {
    print i;
}

i = 0;
while (i < 10) {
    print i;
    i = i + 1;
}
```

Optional Homework

You have all you need now to implement `break` and `continue` statements. These are a fun exercise, and it might be surprising how little code you need to implement them.

I strongly recommend everyone be keeping up to date. I know a lot of students are a week or two behind, that's all fine. But if you haven't written any code yet, I am worried you will not complete the course. So if you want the certification but haven't started programming yet, I would aim to at least have parsing done by the end of the week.