STS Software Technology Systems TUHH

Compiler Construction
Summer Term 24
June 13
Lübke/Schupp

Code Generator

Deadline : 27.06.24 08:00 AM (2024-06-27T08:00:00+02:00)

How to complete a project successfully?

Follow the rules as described in the Lecture!

GitLab at TUHH[1] is used as the submission system in this course. Your GitLab credentials are the same as for StudIP. If you cannot log into GitLab with your StudIP login and password, contact the teaching assistant immediately. All your code must be pushed to your group repository provided to you by the teaching assistant. Remember, that each group member needs to demonstrate their contributions through code commits. You can update your solution until the deadline. The latest commit before the deadline is considered by default. If you want your late submission to be graded for penalized maximal number of possible points, the **whole** group must send an email to the teaching assistant **before** the interview day.

How to get additional information?

You are encouraged to discuss past and present project sheets with the teaching assistants. Either approach the teaching assistant during the exercise session, or visit us during the weekly office hours. We are also available through e-mail or on the StudIP forum. We try to reply as quickly as possible and in general, you should get a reply the next weekday, but we cannot guarantee this.

[1] https://collaborating.tuhh.de/

Code Generation with LLVM

Your task is to compile a small CPP language to LLVM IR. LLVM IR is structured in basic blocks, meaning a sequence of non-branching instructions, terminated by a control-flow-affecting instruction (terminator), e.g., jumps and returns. Furthermore, LLVM IR makes evaluation order explicit. The LLVM IR for the expression $a + b + c$ would, e.g., be compiled to something similar to the following:

```
%1 = add i32 %a, %b
%2 = add i32 %1, %c
```

The LLVM language reference[2] gives you a more precise overview. If you use Haskell as your implementation language, look at the modules `llvm-hs`[3] and `llvm-hs-pure`[4]. If you use C++ you can directly refer to the Kaleidoscope tutorial[5], and the LLVM Programmer's Manual[6]. Regardless of language (although it uses C++), the tutorial by Mukul Rathi[7] was quite helpful to other students in the past.

To check your generated IR, you can execute it using the `lli-9` command[8].

Tasks

Your task is to implement a code generator that takes programs in a simple CPP language, and generates corresponding LLVM IR. The syntax of the language is described in the (enclosed) file *CPP2.cf*. For a description of the semantics, please refer to this project sheet.

Task 1 Implement the base part of the code generator. The base does not include while, do-while, for, and if statements, binary arithmetic operators `+`, `-`, `*`, `/`, and arithmetic comparison operators `<`, `>`, `<=`, `>=`. Work on this task in groups.

Task 2 Look up the file `ASSIGNMENTS` in your GitLab repository and implement the parts of the language corresponding to your number:

(1) do-while-loops, `+`, and `<`

[2] http://llvm.org/docs/LangRef.html
[3] https://hackage.haskell.org/package/llvm-hs
[4] https://hackage.haskell.org/package/llvm-hs-pure
[5] https://releases.llvm.org/9.0.0/docs/tutorial/LangImpl03.html
[6] https://releases.llvm.org/9.0.0/docs/ProgrammersManual.html
[7] https://mukulrathi.com/create-your-own-programming-language/llvm-ir-cpp-api-tutorial/
[8] https://releases.llvm.org/9.0.0/docs/CommandGuide/lli.html

(2) while-loops, `-`, and `>`

(3) for-loops, `*`, and `<=`

(4) if-statements, `/`, and `>=`

Work on your assignments **individually**. The commit with the implementation must be done by the group member assigned to the corresponding language construct.

Summary

The objective of this assignment is to write a code generator from a fragment of the C++ programming language to LLVM, the Low Level Virtual Machine. The code generator should produce LLVM IR. The recommended implementation is via a BNF grammar processed by the BNF Converter (BNFC) tool. The code generator should make a single pass over the code.

Language specification

The language is similar, but smaller compared to the one from Project 2, and you can use the grammar file `CPP2.cf` (*example.cpp2* contains an example program). `int` is the only numeric type, exceptions and structure inheritance are removed, and there is only a single, global scope for symbols. As a consequence, variable declarations must be placed *outside* of functions, and functions can't take any arguments (remember, there are only global symbols/variables/functions!). Additionally, only symbols which have already been defined can be used (to enable single-pass compilation), so the following is **not** possible:

```
int f {
  return g(); // error: g is not defined
}

int g {
  // do stuff;
}
```

Values

There are four types of values:

- integer values, e.g. `42`
- boolean values, `true` and `false`
- a void value
- user-defined structures

Instead of boolean values, you can use (1-bit) integers. Then `true` can be interpreted as `1` and `false` as `0`.

Programs

A program is a sequence of variable, function and structure definitions.

A variable definition consists of the type and name of the variable, e.g.,

```
int i
```

(note the absence of a semicolon, it is not required for variable declarations in this language). It adds the variable to the global environment. If the name already exists in the global environment, the program is ill-formed.

Each function has a body, which is a sequence of statements. The statements in the body are executed in the order defined by their textual order as altered by loops and if conditions. The function returns a value, which is obtained from the return statement. This statement can be assumed to be the last one in the function body. If the return type is void, no return statement is required.

Statements

An expression statement, e.g.

```
i++ ;
```

is evaluated, and its value is ignored. A while statement, e.g.

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

is executed so that the condition expression is first evaluated. If the value is `true`, the body is executed, and the while statement is executed again. If the value is `false`, the statement after the while statement is executed. A do-while statement, e.g.

```
do ++i ; while (i < 10) ;
```

is executed so that the loop body is executed once and then the execution follows the rules for the while statement. A for statement, e.g.

```
for ( i = 0 ; i != 10 ; ++i ) k = k + i ;
```

is executed so that the first expression is evaluated, then the second expression is evaluated and its value is checked. If it is `true`, the body is executed, the third expression is evaluated after it, and the execution repeats starting from evaluation of the second expression. If the value is `false`, the statement after the for statement is executed. An if-else statement, e.g.

```
if (1 < 10) i++ ; else j++ ;
```

is executed so that the condition expression is first evaluated. If the value is `true`, the statement before else is executed. If the value is `false`, the statement after else is executed. A return statement is executed by evaluating its expression argument. The value is returned to the caller of the function, and no more statements in the function body are executed. You can assume that the return statement is always the last one in a function body.

Expressions

The execution of an expression, also called evaluation, returns a value whose type is determined by the type of the expression.

A literal, e.g.

```
123
true
```

is not evaluated further but just converted to the corresponding value. A variable, e.g.

```
x
```

is evaluated by looking up its value in the global context. A projection expression, e.g.

```
e.x
```

evaluates to the value of the field `x` of the value of the expression `e`. A function call, e.g.

```
foo()
```

is evaluated by executing its body. A postincrement,

```
i++
```

has the same value as its body initially has (here `i`). The value of the variable `i` is then incremented by 1. `i--` correspondingly decrements `i` by 1. A preincrement,

```
++i
```

has the same value as `i` plus 1. This incremented value replaces the old value of `i`. The decrement and double variants are analogous. Arithmetic, boolean, and comparison operations,

```
+ a
- a
a   +  b
a   -  b
a   *  b
a   /  b
a <=>  b
a   <  b
a   >  b
a  >=  b
a  <=  b
a  ==  b
a  !=  b
a  &&  b
a  ||  b
```

are interpreted by evaluating their operands from left to right. The resulting values are then added, subtracted, etc. The three-way comparison returns 1, 0, or -1 if `a` is greater than, equal to, or less than `b` respectively. The comparison operators `==` and `!=` return `false` and `true` respectively for operands of different types. Structure comparison is deep, i.e., performed field by field. Assignment,

$$e = a$$

is evaluated by first evaluating `a`. The resulting value is returned, but also the global context is changed by assigning this value to `e`, where `e` is either a variable or a (nested) structure projection of a structure variable.

Format

Input and output

The code generator must be a program called `compiler`, which reads the standard input or a file given as a command line argument.

In case of success, the return value of the program is 0 and it outputs valid LLVM IR corresponding to the input on standard output.

In case of failure, the behavior is undefined (garbage in – garbage out), i.e., the program may output `SYNTAX ERROR` or `TYPE ERROR`, or even `ADKLn235klndsg`. It may also return any value or even just crash.

The easiest way to produce the proper format is to use the (enclosed) ready-made templates.

Success criteria

Your code generator must pass the testsuite available on INGInious[9] accessible from the TUHH network or through the TUHH VPN. You can log in with you usual TUHH user account (the one you also use for StudIP).

After you log in, if you haven't done so already, click the *Course List* and register for the *Compiler Construction* course. When you're registered, click *My Courses* in the side bar, select *Compiler Construction*, and then the *Code Generator* task. Use the *Choose File* button to select the file you want to submit and then click *Submit*. Your submission must be a ZIP archive containing your sources and the grammar file excluding the automatically generated BNFC files and the compiled compiler. The Haskell and C++ submissions must be compilable with `stack build` and `clang++-9` respectively.

When all tests are run you'll see a red or green box (depending on whether your submission already passes all tests). In this box, there are two sections, *debug info* and *test output*. Most of the time, you can ignore the *debug info*, but please always include it when you encounter problems with INGInious and ask for help. The *test output* will tell you how many tests your submission already passed, and gives hints on how to pass any remaining tests.

Compilation together with passing all tests takes a few minutes and may take longer depending on the server load. Please, be patient and do not

[9]https://inginious.sts.tuhh.de

resubmit the same solution to the same problem. If you experience timeouts, please contact the teaching assistant (please include debug info if shown).

Note that the test suite is built in a gradual manner starting from the simplest cases and basic language constructs ending at more complicated ones. The hint shown upon a test failure relates to the language construct in focus and is relevant only assuming that all the previous ones work correctly. The tests for individual subtasks are based on the base, hence we suggest that you start working on Task 2 only after completing Task 1.

Submission

Submit your solution via GitLab[10]. If you have any problems getting the test program to run, or if you think that there is an error in the test suite, contact the teaching assistant.

Grading

You can earn up to 35 points in this project phase.

5 points are awarded for the readability of you code. This includes but is not limited to: sensible naming of variables/functions/..., logical structure (division into modules or sections in the file(s)), length of functions, depth of nesting of functions, avoidance of code duplication, the use of comments, as well as the formatting. In essence: make sure someone else can quickly understand your code without any further explanations.

18 points are awarded for the group's portion of the work (i.e., task 1), split evenly between definitions, statements, and expressions (6 points each).

12 points are awarded for the individual portion of the work (i.e., task 2), split evenly between statements and expressions (6 points each).

[10] `https://collaborating.tuhh.de`