CSC4180 Macro Compiler Construction Assignment 4

Student ID: 120090712

Project structure

```
The structure of my workspace:
.
├── csc4180_A4_120090712.pdf
├── testcases
├── a4.py
├── get_input_ast.sh
├── main.cpp
├── Makefile
├── node.cpp
├── node.hpp
├── parser.y
├── runtime.c
├── scanner.l
└── verify.sh
```

How to execute the compiler

Uplaod the whole file to the cluster and use the following commands:

```
cd path/to/project
bash get_input_ast.sh
bash verify.sh
```

Then you can check the result in /testcases/output/testid.txt...

The IR code generated

I have commented the `print(module)` in `a4.py`, the terminal won't show the ir code. If you want to see it on the terminal, you can uncomment the following code:

```
# Wirte into the llvm_ir
with open(llvm_ir, "w") as f:
    f.write(str(module))
# # print LLVM IR
# print(module) <------------ uncomment it to print the ir code
```

test0:

```
The declaration of the build-in function has been omitted, only the core part is
displayed

define i32 @"main"()
{
entry:
  %"str-2" = alloca [13 x i8]
  store [13 x i8] c"hello world!\00", [13 x i8]* %"str-2"
  %".3" = getelementptr [13 x i8], [13 x i8]* %"str-2", i32 0, i32 0
  call void @"print_string"(i8* %".3")
  ret i32 0
}
```

test1:

```
The declaration of the build-in function has been omitted, only the core part is
displayed

define i32 @"main"()
{
entry:
  %"x-2" = alloca i32
  store i32 10, i32* %"x-2"
  %".3" = load i32, i32* %"x-2"
  %".4" = add i32 0, %".3"
  %".5" = add i32 %".4", 5
  store i32 %".5", i32* %"x-2"
  %".7" = load i32, i32* %"x-2"
  call void @"print_int"(i32 %".7")
  %".9" = load i32, i32* %"x-2"
  ret i32 %".9"
}
```

test2:

```
The declaration of the build-in function has been omitted, only the core part is
displayed

@"x-1" = private constant i32 5
define i32 @"main"()
{
entry:
  %".2" = load i32, i32* @"x-1"
  call void @"print_int"(i32 %".2")
  %"x-2" = alloca i32
  store i32 10, i32* %"x-2"
```

```
  %".5" = load i32, i32* %"x-2"
  call void @"print_int"(i32 %".5")
  %".7" = load i32, i32* %"x-2"
  ret i32 %".7"
}
```

test3:

```
The declaration of the build-in function has been omitted, only the core part is
displayed

define i32 @"main"()
{
entry:
  %"y-2" = alloca i32
  store i32 5, i32* %"y-2"
  %".3" = load i32, i32* %"y-2"
  call void @"print_int"(i32 %".3")
  %".5" = alloca [2 x i8]
  store [2 x i8] c"\0a\00", [2 x i8]* %".5"
  %".7" = getelementptr [2 x i8], [2 x i8]* %".5", i32 0, i32 0
  call void @"print_string"(i8* %".7")
  %".9" = load i32, i32* %"y-2"
  %".10" = icmp sgt i32 %".9", 0
  br i1 %".10", label %"if_block", label %"else_block"
if_block:
  %"is_y_positive-3" = alloca i32
  store i32 1, i32* %"is_y_positive-3"
  %".13" = load i32, i32* %"is_y_positive-3"
  call void @"print_bool"(i32 %".13")
  br label %"merge_block"
else_block:
  %"is_y_positive-4" = alloca i32
  store i32 0, i32* %"is_y_positive-4"
  %".17" = load i32, i32* %"is_y_positive-4"
  call void @"print_bool"(i32 %".17")
  br label %"merge_block"
merge_block:
  ret i32 0
}
```

test4:

```
The declaration of the build-in function has been omitted, only the core part is
displayed

define i32 @"main"()
{
entry:
```

```
  %"i-3" = alloca i32
  store i32 0, i32* %"i-3"
  br label %"for_loop_cond"
for_loop_cond:
  %".4" = load i32, i32* %"i-3"
  %".5" = icmp slt i32 %".4", 10
  br i1 %".5", label %"for_loop_body", label %"for_loop_end"
for_loop_body:
  %".7" = load i32, i32* %"i-3"
  call void @"print_int"(i32 %".7")
  %".9" = alloca [2 x i8]
  store [2 x i8] c"\0a\00", [2 x i8]* %".9"
  %".11" = getelementptr [2 x i8], [2 x i8]* %".9", i32 0, i32 0
  call void @"print_string"(i8* %".11")
  %".13" = load i32, i32* %"i-3"
  %".14" = add i32 0, %".13"
  %".15" = add i32 %".14", 1
  store i32 %".15", i32* %"i-3"
  br label %"for_loop_cond"
for_loop_end:
  ret i32 0
}
```

test5:

```
The declaration of the build-in function has been omitted, only the core part is
displayed

define i32 @"main"()
{
entry:
  %"value-2" = alloca i32
  store i32 10, i32* %"value-2"
  br label %"while_loop_cond"
while_loop_cond:
  %".4" = load i32, i32* %"value-2"
  %".5" = icmp sgt i32 %".4", 0
  br i1 %".5", label %"while_loop_body", label %"while_loop_end"
while_loop_body:
  %".7" = load i32, i32* %"value-2"
  call void @"print_int"(i32 %".7")
  %".9" = alloca [2 x i8]
  store [2 x i8] c"\0a\00", [2 x i8]* %".9"
  %".11" = getelementptr [2 x i8], [2 x i8]* %".9", i32 0, i32 0
  call void @"print_string"(i8* %".11")
  %".13" = load i32, i32* %"value-2"
  %".14" = sub i32 %".13", 1
  store i32 %".14", i32* %"value-2"
  br label %"while_loop_cond"
while_loop_end:
  ret i32 0
}
```

Problem I met and what I have learnt

The condition grammar refers to the rules governing the syntax and semantics of conditions in high-level language constructs, such as if statements and loops. When translating these conditions into LLVM IR, it's essential to handle different types of data, including pointers and arrays.

Challenges Encountered: The primary challenge arose from the distinction between `[size x i8]` and `i8*` types in LLVM IR:

1. `[size x i8]`: This type represents an array of `i8` (8-bit integer) elements with a specific length `x`.
2. `i8*`: This type represents a pointer to an `i8` element, commonly used for representing strings or memory addresses.

Problem Statement: The problem stemmed from the need to handle conditions involving arrays (`[size x i8]`) and pointers (`i8*`) correctly within the LLVM IR generation process. The called function requires a `i8*` string pointer, but the string store in the variable is `[size x i8]`. Specifically, when evaluating conditions or accessing elements within arrays, it was crucial to ensure compatibility and proper type handling.

Approach Taken: To address this problem, I adopted the following approach:

1. **Type Detection:** During the LLVM IR generation process, I implemented logic to detect the types of operands involved in conditions. This included identifying whether an operand was of type `[size x i8]` (array) or `i8*` (pointer).
2. **Type Conversion:** Depending on the detected types, I implemented appropriate type conversions to ensure compatibility and consistency within condition evaluations. For example, when comparing an array with a pointer, I converted the array to a pointer type (`i8*`) before performing the comparison.
3. **Semantic Analysis:** I conducted thorough semantic analysis to validate the correctness of condition expressions and ensure that type conversions were applied consistently and accurately.

Reflections and Learnings:

1. **Deeper Understanding of LLVM IR:** This problem provided valuable insights into the nuances of LLVM IR types and the importance of type compatibility in condition evaluations.
2. **Enhanced Error Handling:** Addressing this problem improved my error-handling capabilities, as I had to account for various scenarios where type mismatches or inconsistencies could occur.
3. **Refinement of Code Generation Techniques:** Through experimentation and iteration, I refined my code generation techniques to handle complex type conversions and ensure robustness in LLVM IR generation.
4. **Understanding of Control Flow Structures:** In addition to resolving the type-related issue, I also gained a deeper understanding of how control flow structures such as if-else statements, while loops, and for loops are implemented in LLVM IR. This understanding was crucial for accurately translating high-level language constructs into LLVM IR.