CS323 Compiler Project Phase 4

SPLCC Compiler Infrastructure Initial Report

Group: 12110804 FANG Jiawei, 12110817 ZHANG Zhanwei, 12110529 CAO Zhezhen

Test Platform

Name	Value
OS	Ubuntu 22.04.2 LTS on Windows 10 x86_64
Bison	GNU Bison 3.8.2
Flex	Flex 2.6.4
libbison-dev	2:3.8.2+dfsg-1build1
libedit-dev	3.1-20210910-1build1
zlib1g-dev	1:1.2.11.dfsg-2ubuntu9.2
llvm-18	1:18$_{++20240106042300+ba3ef331b456-1}$exp1~20240106042415.1415
clang-18	1:18$_{++20240106042300+ba3ef331b456-1}$exp1~20240106042415.1415

Except for the LLVM-related packages, all of them are of default versions available in Ubuntu-22.04.

Project Structure

```
.
├── CMakeLists.txt
├── LICENSE
├── README.md
├── images
|   ├── img-1.png
|   ├── img-2.png
|   ├── img-3.png
|   ├── img-4.png
|   ├── img-5.png
|   └── img-6.png
├── misc
|   └── references
|       ├── syntax.txt
|       └── token.txt
├── modules
|   ├── libspl
|   |   ├── CMakeLists.txt
|   |   └── include
|   |       └── stdargs.h
|   ├── splc
|   |   ├── CMakeLists.txt
|   |   ├── include
|   |   |   ├── AST
|   |   |   |   ├── ASTBase.hh
|   |   |   |   ├── ASTCommons.hh
|   |   |   |   ├── ASTContext.hh
|   |   |   |   ├── ASTContextManager.hh
|   |   |   |   ├── ASTProcess.hh
|   |   |   |   ├── ASTSymbol.hh
|   |   |   |   ├── DerivedAST.hh
|   |   |   |   ├── Expr.hh
|   |   |   |   ├── SPLType.hh
|   |   |   |   ├── SPLTypeContext.hh
|   |   |   |   ├── SymbolEntry.hh
|   |   |   |   ├── TypeCheck.hh
|   |   |   |   └── Value.hh
|   |   |   ├── Analysis
|   |   |   |   └── UnusedVariable.hh
|   |   |   ├── Basic
|   |   |   |   ├── DerivedTypes.hh
|   |   |   |   ├── SPLCContext.hh
|   |   |   |   ├── Specifiers.hh
|   |   |   |   ├── Type.hh
|   |   |   |   └── Typetraits.hh
|   |   |   ├── CodeGen
|   |   |   |   ├── ASTDispatch.hh
|   |   |   |   ├── LLVMWrapper.hh
|   |   |   |   └── ObjBuilder.hh
|   |   |   ├── Core
|   |   |   |   ├── Base.hh
|   |   |   |   ├── Options.hh
|   |   |   |   ├── System.hh
|   |   |   |   ├── Utils
|   |   |   |   |   ├── CommandLineParser.hh
|   |   |   |   |   ├── ControlSequence.hh
|   |   |   |   |   ├── Location.hh
|   |   |   |   |   ├── LocationWrapper.hh
|   |   |   |   |   ├── Logging.hh
|   |   |   |   |   ├── LoggingLevel.hh
|   |   |   |   |   └── TraceType.hh
|   |   |   |   ├── Utils.hh
|   |   |   |   └── splc.hh
|   |   |   ├── IO
|   |   |   |   ├── Driver.hh
|   |   |   |   ├── IOBase.hh
|   |   |   |   ├── Preprocessor.hh
|   |   |   |   └── Scanner.hh
|   |   |   ├── SIR
|   |   |   |   ├── IR.hh
|   |   |   |   ├── IRBase.hh
|   |   |   |   ├── IRBuilder.hh
|   |   |   |   └── IROptimizer.hh
|   |   |   └── Translation
|   |   |       ├── TranslationBase.hh
|   |   |       ├── TranslationContext.hh
|   |   |       ├── TranslationContextManager.hh
|   |   |       ├── TranslationManager.hh
|   |   |       ├── TranslationOption.hh
|   |   |       ├── TranslationUnit.hh
|   |   |       └── TranslationUnitProcess.hh
|   |   └── src
|   |       ├── AST
|   |       |   ├── ASTBase.cc
|   |       |   ├── ASTBasePolymorphism.cc
|   |       |   ├── ASTBaseProcess.cc
|   |       |   ├── ASTBaseType.cc
|   |       |   ├── ASTBaseValue.cc
|   |       |   ├── ASTContext.cc
|   |       |   ├── ASTContextManager.cc
|   |       |   ├── ASTProcess.cc
|   |       |   ├── ASTSymbol.cc
|   |       |   ├── CMakeLists.txt
|   |       |   ├── DerivedAST.cc
|   |       |   ├── Expr.cc
|   |       |   ├── SymbolEntry.cc
|   |       |   ├── TypeCheck.cc
|   |       |   └── Value.cc
|   |       ├── Analysis
|   |       |   ├── CMakeLists.txt
|   |       |   └── UnusedVariable.cc
|   |       ├── Basic
|   |       |   ├── CMakeLists.txt
|   |       |   ├── Type.cc
|   |       |   └── TypeTraits.cc
|   |       ├── CodeGen
|   |       |   ├── ASTDispatch.cc
|   |       |   ├── CMakeLists.txt
|   |       |   └── ObjBuilder.cc
|   |       ├── Core
|   |       |   ├── CMakeLists.txt
|   |       |   ├── Internal.cc
|   |       |   ├── Options.cc
|   |       |   ├── System.cc
|   |       |   ├── Utils
|   |       |   |   ├── CommandLineParser.cc
|   |       |   |   └── Logging.cc
|   |       |   └── Utils.cc
|   |       ├── IO
|   |       |   ├── CMakeLists.txt
|   |       |   ├── Driver.cc
|   |       |   ├── Lexer.ll
|   |       |   ├── Parser.yy
|   |       |   └── Scanner.cc
|   |       ├── SIR
|   |       |   ├── CMakeLists.txt
|   |       |   ├── IR.cc
|   |       |   ├── IRBuilder.cc
|   |        |   └── IROptimizer.cc
|   |        ├── Translation
|   |        |   ├── CMakeLists.txt
|   |        |   ├── TranslationContext.cc
|   |        |   ├── TranslationContextManager.cc
|   |        |   ├── TranslationManager.cc
|   |        |   └── TranslationUnit.cc
|   |        └── splc.cc
|   └── ts
|       └── CMakeLists.txt
└── reports
    ├── 12110804-phase1.md
    ├── 12110804-phase1.pdf
    ├── 12110804-phase2.md
    ├── 12110804-phase2.pdf
    ├── 12110804-phase3.md
    └── 12110804-phase3.pdf

29 directories, 115 files
```

Library Invocation and Encapsulation

The SPL Compiler has been written into pure library form:

- `libSPLCAnalysis`: performing analysis on AST
- `libSPLCAST`: encapsulating:
  - manipulation of AST,
  - Symbols of AST,
  - type checking,
  - value checking,
  - AST polymorphism support,
  - AST processor pass (**transform passes**),
  - AST **Context management** (symbols, linkages, hierarchies)
- `libSPLCBasic`: encapsulating fundamental classes of SPLC, including the singleton-based **Type** system, which supports arbitrary type and their combinations in the entire C programming language, with **pointer-based** type comparison.
- `libSPLCCodeGen`: classes and functions related to **generating object code**. This is achieved by visiting our AST in `ObjBuilder`, calling the **LLVM** system to generate **LLVM IR** code.
  - **Support various target machines**:
    - MIPS
    - Default target machine of the building environment (e.g., AARCH64 for Apple M1)
    - Maybe specified later using **string**.
  - **Supports various optimization passes**.
- `libSPLCCore`: classes and functions related to
  - connecting to the underlying system (**System Wrapper**)
    - Exceptions,
    - Command-Line Parser,
    - Console Output Manager,

    - logging utilities, location tracking utilities
    - **SPLC environment options**
    - ...
- `libSPLCIO`: lexer and parser used during parsing:
    - `SPLCLexer`: Provide pure lexing support based on Flex.
        - **Preprocessor support**
            - `#include`, `#define`, ...
    - `SPLCParser`: Provide **pure parser** (no side-effect on the parse itself) based on GNU Bison.
        - Syntax error analysis
- `libSPLCSIR`: SUSTech IRSIM IR generation.
    - `IRBase`: base classes for IR generation, including `IRStmt`, `IRVar`, `IRFunction`, `IRProgram` and stuff. Exposing abstract interface to the user.
- `libSPLCTranslation`: Providing support for **Translation Units** and **context management during translation process** (any form of translation), including:
    - `TranslationBase`: Provides basic forward declarations, including key types used in lookups during the entire translation process
    - `TranslationContext`: Support for switching translation contexts. This includes:
        - **File Management**, **Macro Management**, **Preprocessor Instructions**
    - `TranslationManager`: Support for managing the entire translation process by providing:
        - **context management**,
            - context lookup/context stack management
        - **variable lookups**,
        - **abstract IO operations**.
    - `TranslationUnit`: Support for a single translation unit, which contains the **AST** and other various translation options
    - `TranslationUnitProcess`: Support for processing translation units.

All of them are in the form of **static library**, such that only static linking needs to be done, and the finally distributed executable requires no additional dynamic library links (though linking to OS' library is required in order to be treated as a valid executable).

To illustrate how these modules are decoupled:

- `libSPLCTranslation`: Allows abstract translation (which may be from source file, IO, or even abstract AST files) to be done without considering the actual lexing/parsing process or even the form of source file. This is done by abstracting away `ASTContext` and `TranslationContext`. It does not know anything about AST except for the definition of `ASTContext`.

Many of the core components relies on `Core` or `Basic` only.

Separated Phase 4 IR Compiler

In Phase 4, we have separated the IR Compiler for MIPS assembly code generation, since SPIM and Mars are not intended to run actual assembly code (which includes linking and relocating instructions, and can be run on OS including but not limited MIPS-GNU-Linux).

SPLCC: Compiling to Object File

Prerequisite

Installing LLVM-18

Following the tutorial on LLVM Debian/Ubuntu packages:

Get the installation script.

```
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 18
```

Next, install `llvm-18` (use `apt` to check installation) and `clang-18`

`sudo apt install llvm-18`

`sudo apt install clang-18`

You might want to make a symbolic link to link `clang-18` to `/usr/bin/clang` (and also for other applications including `llvm-config`).

Installing other requires libraries

On Ubuntu-22.04 (this may not work for other distributions). **Assume you have already added llvm source**. If not, refer to the previous section about installing LLVM-18.

```
sudo apt install -y flex libbison-dev \
                    libedit-dev zlib1g-dev\
                    llvm-18 clang-18
```

If you want to cross compile to MIPS platform *on platform other than MIPS*, you may also need to install the cross-compilation tool chain. You may install:

```
sudo apt install -y gcc-mips-linux-gnu
```

Compiling SPLC

Under the root directory:

```
mkdir -p build && cd build
```

```
cmake --no-warn-unused-cli\
      -DCMAKE_BUILD_TYPE:STRING=Debug\
      -DCMAKE_EXPORT_COMPILE_COMMANDS:BOOL=TRUE\
      -S../ -B./
```

Building the **Release** version will significantly reduce execution time, since in the **Debug** version, ASAN (address sanitizer) is enabled by default.

Then use

```
make
```

to make all SPLCC libraries and the **splc executable**. By default, the **splc** executable will be located under `build/bin/splc`.

Compiling Code to Object File

Get Familiar with the Command-Line Interface

Make sure you have already **compiled** the **splc** executable.

You may view all available commands by typing:

```
user@ubuntu:~/SPLC$ build/bin/splc
usage: splc [-h] [--h] [--genasm] [--target TARGET] SOURCE_FILE
```

in which:

- `--genasm` generates **assembly instead of object file** for input source code under the same directory, with extension ".asm" appended directly to the input file.
- `--target` specifies the target triple to generate. In our case, you may specify:
  - `--target=mips` to **generate MIPS object file**.
  - `--target=default` to generate **default object file that can be run on splc's host machine**.
  - Other targets are not supported as of now.

Example Compilation

Assume we have two files (they are provided under the root directory of the uploaded folder):

- `main.c`: including the main function to be executed, along with other utility functions.

  ```
  extern int write(int k);
  ```

```c
int hanoi(int n, int p1, int p2, int p3);

int main(){
    int sum = 3;
    hanoi(sum, 1, 2, 3);
    return 0;
}

int hanoi(int n, int p1, int p2, int p3){
    if(n == 1){
        write(p1*10000+p3);
    }
    else{
        hanoi(n-1,p1,p3,p2);
        write(p1*10000+p3);
        hanoi(n-1,p2,p1,p3);
    }
    return 0;
}
```

Here we declare `extern int write(int k)` which we will use the library function `int printf(const char *fmt, ...)` to implement. (**Because inline assembly is not supported yet**).

Note that **splc** supports **(extern) forward declaration**. Other features that is used to support object linkage is also implemented.

- `lib.c`: including **IO/Standard Library functions** that needs to be linked to system library.

```c
#include <stdio.h>

extern int write(int k)
{
    printf("%d\n", k);
    return 0;
}
```

First, we use the `splc` **executable** to compile `main.c` to object file, which will later get linked with `lib.c`.

```
user@ubuntu:~/SPLC$ build/bin/splc test.c
main.c:1.12-1.24: debug: the type of the identifier is: function (si32) returning
si32
       1 | extern int write(int k);
         |            ^~~~~~~~~~~~
...
                  `-UIntLiteral <18.12-18.13> , val: 0

splc: debug:
ASTContextTable [0] at 0x60d000004490, size 3
  "write", Func, function (si32) returning si32, declared, at main.c:1.12-1.24
  "hanoi", Func, function (si32, si32, si32, si32) returning si32, defined, at
main.c:3.5-3.41
  "main", Func, function returning si32, defined, at main.c:15.5-15.11
```

```
  ASTContextTable [1] at 0x60d000004560, size 0
  ASTContextTable [1] at 0x60d000004630, size 4
    "n", Param, si32, declared, at main.c:3.11-3.16
    "p1", Param, si32, declared, at main.c:3.18-3.24
    "p2", Param, si32, declared, at main.c:3.26-3.32
    "p3", Param, si32, declared, at main.c:3.34-3.40
    ASTContextTable [2] at 0x60d000004700, size 0
      ASTContextTable [3] at 0x60d0000047d0, size 0
      ASTContextTable [3] at 0x60d0000048a0, size 0
  ASTContextTable [1] at 0x60d000004970, size 0
    ASTContextTable [2] at 0x60d000004a40, size 1
      "sum", Var, si32, defined, at main.c:16.9-16.16

splc: info: writing object file for platform: x86_64-pc-linux-gnu
splc: info: wrote main.c.o
```

Here we note that `int write(int c)` is only **declared** but **not defined**. This declaration is necessary, since it tells splc that **there is a function prototype**. If not, splc will report that symbol has not been defined and throw an error.

After compilation, splc will inform that the object file has been assembled and written into file `main.c.o`. To verify that this is indeed an object file, you may use **objdump** to view the original content. To disassemble, use

```
user@ubuntu:~/SPLC$ objdump -D main.c.o

main.c.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   50                      push   %rax
   ...
  25:   c3                      ret
  26:   66 2e 0f 1f 84 00 00    cs nopw 0x0(%rax,%rax,1)
  2d:   00 00 00

0000000000000030 <hanoi>:
  30:   48 83 ec 18             sub    $0x18,%rsp
  ...
  9d:   48 83 c4 18             add    $0x18,%rsp
  a1:   c3                      ret

Disassembly of section .eh_frame:

0000000000000000 <.eh_frame>:
   0:   14 00                   adc    $0x0,%al
   ...
  44:   02 6d 0e                add    0xe(%rbp),%ch
  47:   08                      .byte 0x8
```

Above is an example of disassembling on an `x86_64` machine.

Next, we compile the library code. Since **splc** does not implement various feature test macros and builtin **vararg** functions (though supported in AST), we compile the library using the default compiler. On Ubuntu, this is `gcc`:

```
user@ubuntu:~/SPLC$ gcc -O3 -Wall -c lib.c
```

In which we compile `lib.c` to `lib.o` for the **IO** function we shall use later. Now, we link them by invoking the default compiler on Ubuntu. Note that it is **not possible to link to system library within splc**.

```
user@ubuntu:~/SPLC$ gcc lib.o main.c.o -Wall
```

Now we may run it!

```
user@ubuntu:~/SPLC$ ./a.out
10003
10002
30002
10003
20001
20003
10003
```

A few other examples has been provided under the folder `./test/llvm-test`. You may try to compile them. Each of them have been tested under x86_64 platform and has been proved to be working.

The intermediate LLVM IR files are also available in the form `{filename}.ll`. They are generated simultaneously under the same source directory. You may use LLVM's JIT to run them.

SPLCC: Implementation

LLVM IR Generation

Implementation of **Object File Generation** is placed under `modules/splc/include/CodeGen` and `modules/splc/src/CodeGen`, in which

- `ObjBuilder`: Support for object file generation, including `AST` visitor and various context managements.
  - `ObjBuilder` starts translation by tracking the root **translation unit**, in which it dispatches the translation process according to the type of the given AST node.
  - For each type (types for vectors and arrays) stored in AST node (stored in the type system `SPLCContext&`) which is initialized in constructor, `ObjBuilder` finds the corresponding type inside `LLVM` through cached type analysis and recursive-descent type parsing.
    - Types are computed using `computeLangType()` during parsing and registered inside `ASTContextManager` through `TranslationManager`. Variables of corresponding type will be stored inside `ASTContext`.

        - In `ASTContext`, a hierarchy is maintained such that user may lock its parent `ASTContext` to search for outer variables, defined struct/union types, and more.
    - For each function, `ObjBuilder` registers a corresponding function in `IRBuilder`. `IRBuilder` maintains a list of functions and their signature in the form `(ReturnType, Arg1Type, ...)`.
    - **Blocks**: Instructions are stored inside `Block`s. A `Block` is a basic construct inside the control flow graph and stores instructions contained in the same order as that in sequential execution. By specifying blocks, and set `IRBuilder` to insert instructions to blocks, we **build the CFG and IR file** at the same time, facilitating the analysis part later.
        - Selection statements, including if/else/switch,
        - Iteration statements, including while/doWhile/For,
        - ...

      can all be supported by simply creating new blocks and inserting blocks.
    - **Restrictions**: Blocks are required to have **one and only one terminator** to allow easier CFG manipulation. A **terminator** is defined to be a jump/branch if/return instruction which causes the control flow to transfer to a different block. The particular case that the C programming language does allow empty return statements causes the return statement to allow NULL supplied as its argument.
- `LLVMWrapper`: Provides header files and necessary facilities for `ObjBuilder`.

Phase 4: Compiling to MIPS instructions for SPIM/MARS

Register Allocation

Given that IR instructions need at most three addresses, a straightforward strategy was chosen: assigning these addresses directly to registers $t0to$t2. This approach simplifies the process by loading the necessary values into these registers for each instruction and then storing them back to memory after execution.

Variable Management

1. **Node Structure**: The `Node` structure is designed to efficiently manage variables in the compiler. Each `Node` includes:
    - `key`: A string for the variable's name.
    - `offset`: An integer indicating the variable's memory location.
    - `reg`: A type representing the register where the variable is stored.
2. `varList`: Maps each variable to its corresponding register and memory address. It's a key component in the project for managing variable storage.
3. `argList`: Store the arguments inside a function.
4. **Functionality**:
    - `insertLast`: This function adds a new variable to the end of the list, used for general variable management.
    - `insertParam`: Specifically used for inserting variables into the parameter list.

- `findRegister`: Retrieves the register associated with a given variable, useful for register allocation tasks.
- `findOffset`: Finds the memory offset of a specified variable, aiding in memory management.

Condition Translation

In TAC, we have defined six kinds of conditional statements, respectively IFLT,IFLE,IFGT,IFGE,IFNE,IFEQ. Each of these TAC (Three Address Code) conditional statements is emitted in a similar manner. To avoid repetitive coding, a macro definition is used to simplify this process.

```
#define EMIT_COND_BRANCH(TAC, MIPS_INST)                                   \
  do {                                                                     \
    Register x, y;                                                         \
    if (_tac_quadruple(TAC).c1->kind == OP_CONSTANT) {                     \
      x = t3;                                                              \
      _mips_iprintf("li %s, %d", _reg_name(x),                             \
                    _tac_quadruple(TAC).c1->int_val);                      \
    } else {                                                               \
      x = get_register(_tac_quadruple(TAC).c1);                            \
    }                                                                      \
    if (_tac_quadruple(TAC).c2->kind == OP_CONSTANT) {                     \
      y = t4;                                                              \
      _mips_iprintf("li %s, %d", _reg_name(y),                             \
                    _tac_quadruple(TAC).c2->int_val);                      \
    } else {                                                               \
      y = get_register(_tac_quadruple(TAC).c2);                            \
    }                                                                      \
    _mips_iprintf(MIPS_INST " %s, %s, label%d", _reg_name(x), _reg_name(y),   \
                  _tac_quadruple(TAC).labelno->int_val);                   \
    restore_regs();                                                        \
  } while (0)
```