Cutting Edsg|er Compiler

Kallas Konstantinos 03112057
Tzinis Efthymios 03112007

November 8, 2016

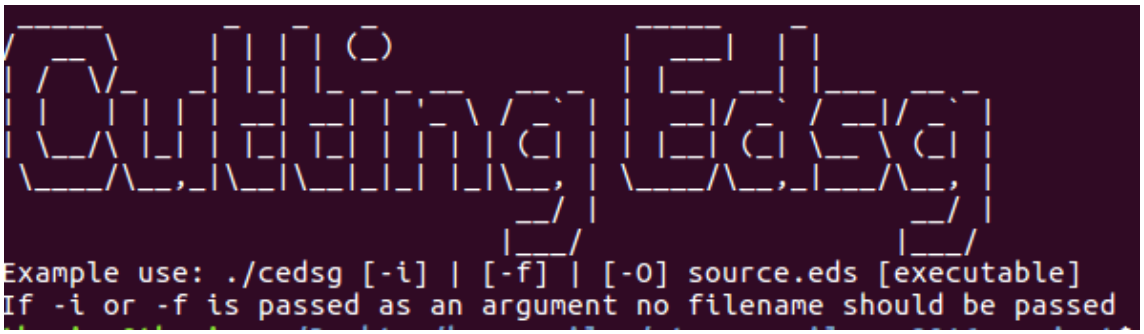

Figure 1: Compiler Options

1 Introduction

In our work we provide an Edsger Compiler according to the specifications given by the leaflet provided by the *Compilers* class 2016-2017 NTUA. We endorsed a more modular solution divided into 5 different sections of lexer, parser, semantic analysis and the LLVM tool for intermediate code production and the final optimized executable binary file. We tried to use the LLVM tool in order to produce an contemporary compiler project which creates a fully optimized code for the x64 architecture. Our source code are mainly comprised by python scripts, some C source was coded in order to take advantage of some run time features and some bash files were written in order to cope with various purposes. Deviant sections are described briefly below with cross reference to the corresponding source files and after that we refer to some key features of our implementation policies and some challenges we faced and tried to solve.

2 Sections

2.1 Lexer

We used the **ply 3.9** Python lex for this section which can be found in https://pypi.python.org/pypi/ply. The corresponding source file in the source folder is **edsger_lexer.py**. This section was quite straight forward to implement and we actually separated from the main text all the tokens and classified them to the respective category.

2.2 Parser

We used the same as above Python yacc which uses LALR(1) parsing which is efficient and well suited for larger grammars. The corresponding source file in the source folder is **edsger_parser.py**. We configured the basic grammar rules for every different token which is inside the docstrings. Every one of the corresponding modules implement some other parser functionality as well (type checking, check if a variable was declared, check if a "break" is included inside a loop, etc.) Moreover, we implemented a hierarchical precedence for all the operators.

2.3 Semantic Analysis

The prominent idea of the implementation was the creation of an AST as an abstract class which can be accessed universally from all the other code files. Each node of this tree represents a different type and embodies subordinate functions or methods which operate on the class itself. The corresponding source file in the source folder is **edsger_semantic_classes.py** although most of the code is used for calling properly the llvm tool.

2.4 LLVM IR

The corresponding source file in the source folder is **edsger_ir.py** which is essential to the IR Generation. The file **edsger_ir_tester.py** used for early development purposes and it is no longer used in our project but it is left there as a reminiscent of our early approach in the project.

2.5 Other files

- **parsetab.py** and **parser.out** are created automatically by our lexer.
- **warning_messages.py** contains wrapper error functions in order to separate message designing from developing.
- **obj** folder contains all the run time statically linked C libraries we produced for the new and delete functionality and the manipulation of doubles. In the source folder **thy_kos_new_delete.c** and **thy_kos_strold.c** are the source files for these libraries respectively.
- The folder **tests** contains some testcases and the expected output of them for our compiler we used to check its validity in deviant functionality.
- **Makefile** follows a very easy to follow and use structure providing an extra target of *tests* which runs all the test provided in the respective folder.
- Considering that a potential test case is included in the test files we can use the bash script **run_ir_test.sh** by providing only the name of the file.

$$sh\ run\_ir\_test.sh\ <filename>$$

- Also we could actually use the script **run_tests.py** with the file **tests_to_run** which has all the names from the test cases in the respective older we need to run. Every line of the later file has the following form: $<name>\ \ in$ if input is required or $<name>$ otherwise.
- **cedsg** is the file the user has to run for testing our compiler, in 1 the reader could take a look in the help menu of our compiler.

3 Challenges

In this section we will briefly describe all the non trivial challenges we faced and the way we approached them.

3.1 #include statements

Challenge: A library can include itself which potentially leads to an infinite loop of includes.
Approach: In order to avoid this problem we keep included libraries in a list not allowing "re-includes". We then lex, and parse each different library to a different AST and then generate code for the libraries first and then for the subject source AST.

3.2 Calling functions with many arguments (Comma separator)

Challenge: The parser recognized the commas in function calls as the binary operator comma.
Approach: We decided to change the parser rules so that the function call rule recursively "ingests" its arguments.

3.3 Function Naming

Challenge: In edsger there can be functions with the same name and different arguments in the same scope level and functions with the same name and argument in different scope levels. However in LLVM IR functions are global and there can only be one function with the same name.
Approach: We extended the names of the functions in llvm ir with argument and scope information, so that there is no name conflict in LLVM IR.

3.4 Arrays and L-Values

Challenges: Arrays are not considered L-Values. The following should not be allowed.

```
// Invalid
char a[5];
a = "foo\n";
```

Approach: We implemented this check in the semantic analysis.

3.5 Constant String Immutability

Challenges: Constant strings should be immutable. *Approach:* We solved this problem by declaring all string constants as immutable globals, and then referencing them throughout the program by pointers. A constant string that appears more than one time in the text will be declared only once and referenced all the times. This way we not only have immutable string constants, but we also achieve better memory usage.

3.6 Nested Functions

Challenges: Nested Functions have access to the outer functions' variables and parameters. However LLVM IR only allows global functions. *Approach:* We solved this problem by encapsulating all the variables and parameters of each function in a struct, and then pass this struct as the final argument in each non global functions.

3.7 Runtime Checks

Challenges: There was need for two runtime checks. One before calling the delete function and one before comparing two pointers. *Approach:* We needed to create some functions to perform those runtime checks. We decided to write them in C, for ease of implementation, and then compile them in a static library that will be linked with the object file.

3.8 Real Numbers

Challenges: The edsger specifications, required 80 bit floating point numbers. The Python API we used for the code generation stage doesnt support fp80 numbers. Python doesnt also support fp80 numbers in general. So we had to create the support ourselves.
Approach: We first created the needed classes (That we are planning to merge to the Python API we used). We also created a dynamic C library in order to convert string constants to the correct constant representation for LLVM IR.
Note: We were careful to represent the float constants in python only as string because otherwise we would lose some precision to 64 bits.