CS420 Compiler Design
Report for the Term Project: Final Report

Team 12

Jaeseong Choe
Undergraduate
Department of Physics, KAIST

Kee Tack Kim
Undergraduate
Department of Mathematics, KAIST

Taeyoung Kim
Undergraduate
School of Computing, KAIST

Youngrae Kim
Undergraduate
School of Computing, KAIST

Seokbin Lee
Undergraduate
School of Computing, KAIST

December 26, 2019

1 Introduction

1.1 ARTIDE

We named our project by *ARTIDE*. The word ARTIDE is an abbreviation for *A Really Tiny Integrated Development Environment*. ARTIDE provide only compiler and debugger for the mini-C programming language. Since it does not provide other common options of IDE like source code editor and linker, it seem to deficiency to called by IDE. However, we choose this naming for the future development. In Italian, the word ARTIDE means the north pole of the earth. Hence, we choose polar bear as our trademark. The project ARTIDE implemented by using Python programming language.

Figure 1: Polar bears.

1.2 Directory structure

Our project directory has the following structure:

```
/
├── /doc
│   ├── /doc/img
│   ├── /doc/tex
│   ├── Some documentations from TA
│   └── Some documentations by ourselves
├── /lib
│   └── /lib/ply
│       ├── /lib/ply/lex.py
│       └── /lib/ply/yac.py
├── /src
│   ├── /src/lexical_analyzer.py
│   ├── /src/syntax_analyzer.py
│   ├── /src/semantic_analyzer.py
│   ├── /src/intermediate_code_generator.py
│   ├── /src/code_generator.py
│   └── /src/debugger.py
├── /test
├── /.gitignore
├── /LICENSE
└── /README.md
```

In the root directory `/`, there are four sub-directories `/doc` for store the documentation files, `/lib` for library files, `/src` for source code files, and `/test` for test code files. In the root directory, there are also some files that containing some information of the project. `/.gitignore` contains the ignore information for git, `/LICENSE` contains the license information of the project, and `/README.md` contains explanation about the project as the form of markup document.

1.3 Publication

We published our project as GitHub public repository with MIT License. You can check our source code and other information from https://github.com/JaeseongChoe/KAIST-CS420-Term_Project.

2 mini-C specification

The mini-C programming language is an subset of the ANSI-C (C89/C90).
The mini-C supports:

- Primitive data types:

  `int`, `float`, `double`, `char`, `str` + `array` and `pointer` types for them

- Primitive operations:

  Arithmetic, comparison&relation, logical, bitwise, and assignment operations.

The mini-C does not supports:

- Some complex data types:

  `struct`, `union`, and `enum` types.

- User defined data types.

- Type qualifiers:

  `signed`, `unsigned`, `const`, `volatile`, `static`, `auto`, and `register`.

3 Libraries and modules

3.1 PLY library

Lexical analyzer and syntax analyzer of the mini-C compiler in ARTIDE implemented by using PLY (Python Lex-Yacc) library (PLY-3.11). The PLY library has two modules `lex.py` and `yacc.py`. In the `lex.py` file, there is a special class for tokenization phase called `LexToken`. The class `LexToken` has four attributes:

- `self.type`

  `self.type` field represent the type of each token. For example, lexime `1234` has the token type `ICONST` after it tokenized.

- `self.value`

  `self.value` field represent the original string of each token. For example, lexime `1234` has the value `‘1234’` after it tokenized.

- `self.linno`

  `self.lino` field represent the line number of each lexime in the source file.

- `self.lexpos`

  `self.lexpos` field represent the position of first charactor of each lexime relative to the start of source file.

Figure 2 shows that how the tokenization phase works. If the lexical analyzer meet `var = 1234;` at line 10 of source file, then it produces the tokens like `(ID, ‘var’, 10, 53)`, `(ASSIGN, ‘=’, 10, 57)`, `(ICONST, ‘1234’, 10, 59)`, and `(COLON, ‘:’, 10, 63)` with predefined matching rules for each lexime.

```
 9 ···
10 var = 1234;
11 ···
```

$\longrightarrow$

```
...
(ID, ‘var’, 10, 53)
(ASSIGN, ‘=’, 10, 57)
(ICONST, ‘1234’, 10, 59)
(COLON, ‘:’, 10, 63)
...
```

Figure 2: Tokenization.

3.2 `node.py` module

The `node.py` module is an our own module for constructing the abstract syntax tree of the input `.c` file. There is only one class called `Node` in this module and it has the following attribute:

- `self.type`

  The field `self.type` represent the type of the node. This type information may can be `INT`, `CHAR`, `If`, `IfElse`, `While`, and `For`.

- `self.value`

  The field `self.value` represent the label of the node. It can be operator like `+` and `-` for some expression statement.

- `self.lineno`

  The field `self.lineno` represent the line number of the statement in the original input `.c` file.

- `self.children`

  The field `self.children` represent the list of child nodes.

3.3 `ast.py` module

The `ast.py` module is an our own module for construction the abstract syntax tree of the input `.c` file with another scheme. In `ast.py` module, there are many class with can be divide into three groups. The first group for defining type information, the second group for abstract syntax information of the operators, and the third group for defining various types of node in abstract syntax tree.

The first group has only one class `Type(enum.Enum)`. It defines the type information like `Type.VOID`, `Type.INT`, `Type.FLOAT`, `Type.CHAR`. This type information used for translation from concrete syntax to abstract syntax, and implementation of the type checker.

The second group has classes that define the abstract syntax information of operators. The role of it is removing ambiguity in the concrete syntax. For example, the token `+` in concrete syntax can be has two different semantic, namely, binary addition operator `+` (eg. `x + y`) and unary sign operator `+` (eg. `+3.14`). The second group has the following list of classes `Operator(enum.Enum)`, `ArithOp(enum.Enum)`, `ComRelOp(enum.Enum)`, `LogicalOp(enum.Enum)`, `BitwiseOp(enum.Enum)`, `AssignOp(enum.Enum)`, and `MemPoinOp(enum.Enum)`.

The third group has classes that define various types of node in abstract syntax tree. It contains `Node(object)`, `ID(Node)`, `Subscript(Node)`, `FunCall(Node)`, `Args(Node)`, `UnaOp(Node)`, `BinOp(Node)`, `TerOp(Node)`, `Assign(Node)`, `Expr(Node)`, `If(Node)`, `IfElse(Node)`, `Switch(Node)`, `Case(Node)`, etc.

3.4 `symtab.py` module

The `symtab.py` module is an our own module for construct the symbol table and funtion table. It implemented by using the dictionary which is the one of Python built-in data type to provide hash table feature. Our symbol table structure has the hierarchy as shown in Figure 3 by its scope (or block). In order to implement this hierarchy, we define the two classes `SymTabBlock` and `SymTab`. The class `SymTabBlock` represents a symbol table for certain block, and `SymTab` represents and manages a overall hierarchy of these blockwise symbol tables.

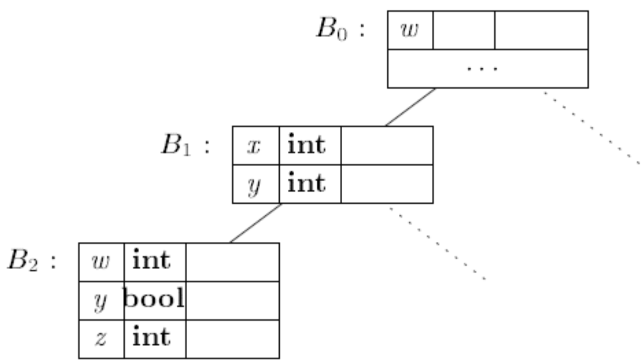

Figure 3: The hierarchy of symbol table. [1]

Furthermore, there is one another class for collecting the data about certain symbol (or identifier) called `SymTabEntry`. The class `SymTabEntry` has three attributes:

- `self.id`

  `self.id` field represents the identifier as a string.

- `self.type`

  `self.type` field represents the type of that identifier. For example, if the identifier `var` declared in source file with the type `int`, then `self.type` of `SymTabEntry` for this identifier is `INT`.

- `self.assigned`

  `self.assigned` field represents the weather some value was assigned to that identifier or not by boolean (`True` or `False`).

`SymTabBlock` is a symbol table of each block. The class `SymTabBlock` has three attributes:

- `self.prev`

  `self.prev` field represents the symbol table of the direct outer block of current block. For example, in the case of Figure 3, `self.prev` of B_1 pointing to B_0.

- `self.nexts`

  `self.nexts` field represent the list of symbol tables for the direct inner blocks of current block. For example, in the case of Figure 3, `self.nexts` of B_1 is a list `[`B_2`, ...]`.

- `self.table`

  `self.table` is an actual table to save the information about each symbols. It is a dictionary object which is a builtin hash table object in Python. Hence, it corresponds to table in the right side of each labels in Figure 3.

`SymTab` is a management system for overall collection of `SymTabBlock`. The class `SymTab` has one attribute and five methods:

- `self.cur`

  `self.cur` pointing to the current symbol table.

- `insert_block_table(self, block_table)`

  `insert_block_table` method provides feature that insert new symbol table into the management system. It appends the `block_table` into the list `self.nexts` of the current symbol table. Then, it changes `self.cur` to `block_table`.

- `remove_block_table(self)`

  `remove_block_table` method provides feature that remove the current symbol table. It pops the current symbol table from the list `self.nexts` of symbol table for direct outer block. Then, it changes `self.cur` to symbol table for direct outer block.

- `insert(self, symbol)`

  `insert` method provides a feature that register the information of new symbol into current symbol table. The input parameter `symbol` is a `SymTabEntry` object. `insert` method check that there already exist a symbol with same identifier with the input parameter `symbol`. If there is no such a symbol, then `insert` registers the `symbol` into current symbol table with setting the hash key as its identifier `symbol.id`. If there is such a symbol, `insert` produces an error `DupDeclError`.

- `remove(self, id)`

  `remove` method provides a feature that deregister the information about identifier `id` from the current symbol table. `remove` check weather there exist the information about that identifier. If there is such information, then it removes the hash information of that identifier. If there no such information, then it produces an error `UndefIdError`.

- `get(self, id)`

  `get` method provides a feature that searching the information about identifier `id`. `get` searches the information with the manner of starting from current symbol table to outer symbol tables. If `get` succeed to find that information, then it returns that information as the form of `SymTabEntry`. If `get` failed to find that information, then it produces an error `UndefIdError`.

Figure 4 illustrates the hierarchy of symbol table. The outermost rectangle of blue color represents the symbol table management class `SymTab`. An arrow represents the `self.cur` attribute of `SymTab`. The rest rectangles of gray color represent the symbol tables class `SymTabBlock` for each blocks.

4 mini-C compiler

The mini-C compiler in ARTIDE has three different parts, namely, front end, middle end, and back end. The front end consist of three phases lexical, syntax, and semantic analysis. The middle end and back end both consist of only one phase intermediate code generation and code generation, respectively.

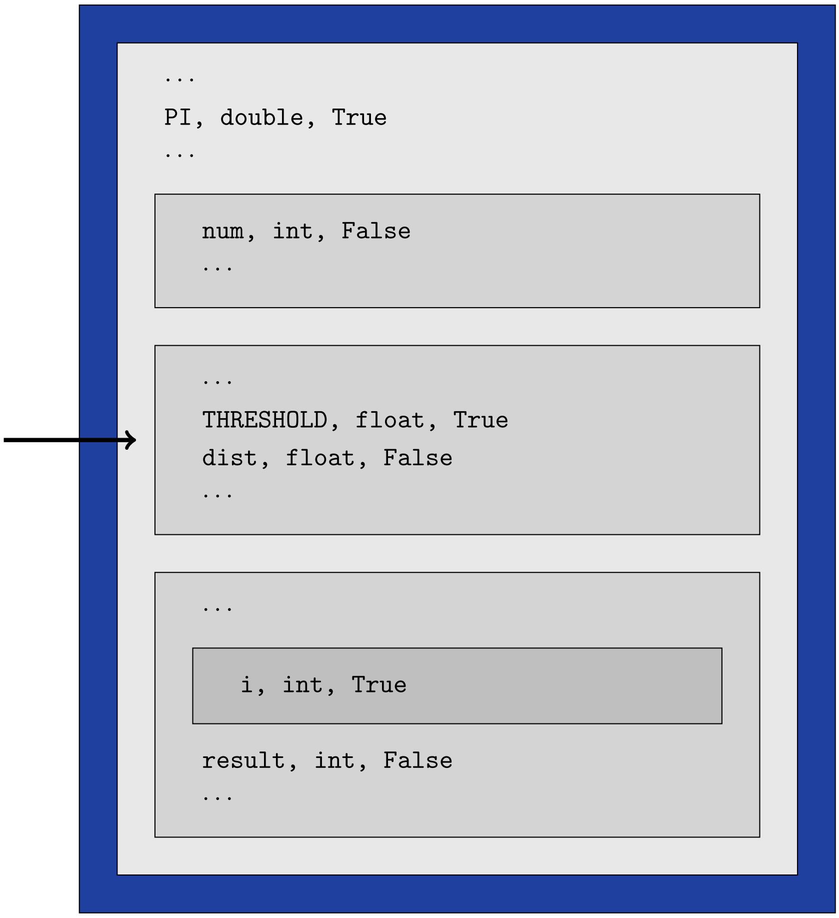

Figure 4: Illustration of the hierarchy of symbol table classes.

4.1 Lexical analyzer

The lexical analyzer uses `lex.py` module of the PLY library. Token specification of mini-C can be found in `/src/lexical_analyzer.py`. It can covers all the tokens of ANSI-C (C89/C90).

4.2 Syntax analyzer

The syntax analyzer uses `yacc.py` module of the PLY library. Context sensitive grammar within BNF can be found in `/src/syntax_analyzer.py`. The parsing mechnism of generated syntax analyzer is LALR(1).

4.3 Semantic analyzer

The semantic analyzer of the mini-C compiler in ARTIDE provides three different feature.

- Construction of the symbol table and the function table.
- Static checking
  - While the constructing the symbol table and the function table, it check the existence of undeclared or duplicated identifier error.
  - It also check that weather the `continue` and `break` statement are located inside loop or not.
- Type checking
  - Check the types of operands for arithmetic operators.
  - Check the types of destination and source element in assignment statement.
  - While the type checking phase, insert the proper type casting operation when it need to and possible to match the types of operands.

4.4 Intermediate code generator

The intermediate code generator get the abstract syntax tree as its input and returns the string of intermediate code as its output. The form of intermediate code is the *three address code*.

4.5 Code generator

The code generator get the string of intermediate code as its input and returns the string of assembly code for the x86 architecture with AT&T syntax.

5 mini-C debugger

The mini-C debugger in ARTIDE provide two different features, namely, interpreting feature and debugging feature.

5.1 Interpreting feature

Interpreting feature provides the way that interpret the input `.c` file as line by line. There are only one command to control the interpreting feature.

- `next`

  The command `next` excutes a single or multiple lines(s) of the source code. For example. `next` just excutes current line of source code, and `next 10` will excute 10 lines including current line.

5.2 debugging feature

Debugging feature provides the helpful options for debug the input `.c` file. There are two commands to control the debugging feature.

- `print`

  The command `print` provides feature that print the value contained in a variable at the moment. For example, if an integer variable `x` contains value `10`, then `print x` will print `10`.

- `trace`

  The command `trace` provides feature that trace the history of a variable from the beginning to the moment.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Jaeseong Choe, Kee Tack Kim, Taeyoung Kim, Youngrae Kim, and Seokbin Lee. *Team 12, CS420 Term Project (Final Presentation)*. KAIST. Dec. 2019.

[3] Jaeseong Choe, Kee Tack Kim, Taeyoung Kim, Youngrae Kim, and Seokbin Lee. *Team 12, CS420 Term Project (Management Plan)*. KAIST. Oct. 2019.

[4] Jaeseong Choe, Kee Tack Kim, Taeyoung Kim, Youngrae Kim, and Seokbin Lee. *Team 12, Internal Data Structure*. KAIST. Nov. 2019.

[5] Kyuho Son. *Term Project: Final Presentation*. KAIST. Dec. 2019.

[6] Kyuho Son. *Term Project: Internal Data Structure*. KAIST. Nov. 2019.

[7] Kyuho Son. *Term Project: Interpreter Implementation*. KAIST. Oct. 2019.

[8] Kyuho Son. *Term Project: Optional Feature 1. Memory Management*. KAIST. Oct. 2019.

[9] Kyuho Son. *Term Project: Optional Feature 2. Recursive Function Call*. KAIST. Oct. 2019.

[10] Kyuho Son. *Term Project: Optional Feature 3. Code Generation*. KAIST. Oct. 2019.

[11] Kyuho Son. *Term Project: Optional Feature 4. Code Optimization*. KAIST. Oct. 2019.

[12] Kyuho Son. *Term Project: Optional Feature 5. Compile Error Handling*. KAIST. Oct. 2019.

[13] Kyuho Son. *Term Project: Presentation on the Plan*. KAIST. Oct. 2019.

[14] Kyuho Son. *Term Project: TA Session*. KAIST. Oct. 2019.