

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.



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`.

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 this it is removing ambiguity in the concrete syntax. For example, the token `+` can be two different semantic in the concrete syntax, 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 2 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.

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.

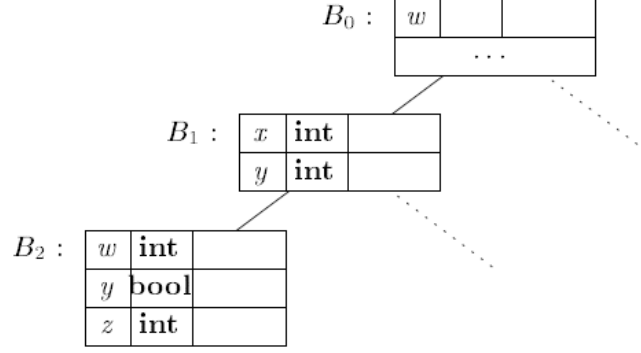


Figure 2: The hierarchy of symbol table. [?]

- **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 2, **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 2, **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 2.

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 3 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.

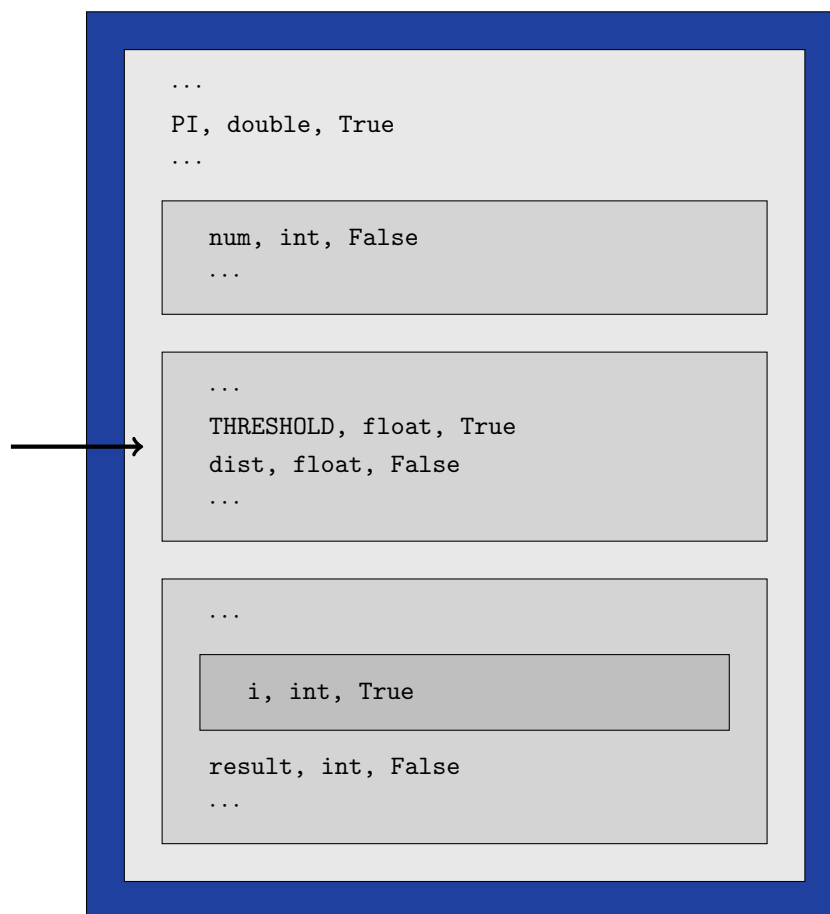


Figure 3: Illustration of the hierarchy of symbol table classes.

4 mini-C compiler

4.1 Lexical analyzer

4.2 Syntax analyzer

4.3 Semantic analyzer

4.4 Intermediate code generator

4.5 Code generator

5 mini-C debugger