CS420 Compiler Design
Report for the Term Project: Internal Data Structure

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

November 21, 2019

1 Token

The PLY library consist of two `.py` files. `lex.py` for lexical analyzer generator and `yacc.py` for syntax analyzer generator, respectively. 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 1 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 ...
```

→

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

Figure 1: Tokenization.

2 Abstract syntax tree

In order to implement the abstract syntax tree (AST), we implement the class `Node`. The class `Node` has two attribute:

- `self.data`

  `self.data` is a field to represent general data contained in that node. The type of data can be anything by the beauty of Python!

- `self.children`

  `self.children` is a field that represent the children of that node. The type of it is a list.

Figure 2 shows that the graphical explanation of the class `Node`.

```
Node

self.data = anything
self.children = [child1, child2, ...]
```

Figure 2: Class `Node`.

3 Symbol table

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.

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`).

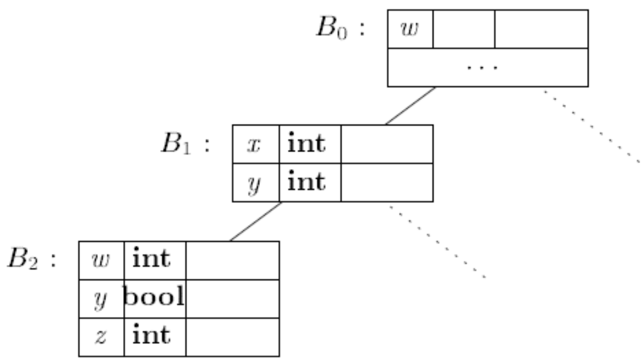

Figure 3: The hierarchy of symbol table. [1]

`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.

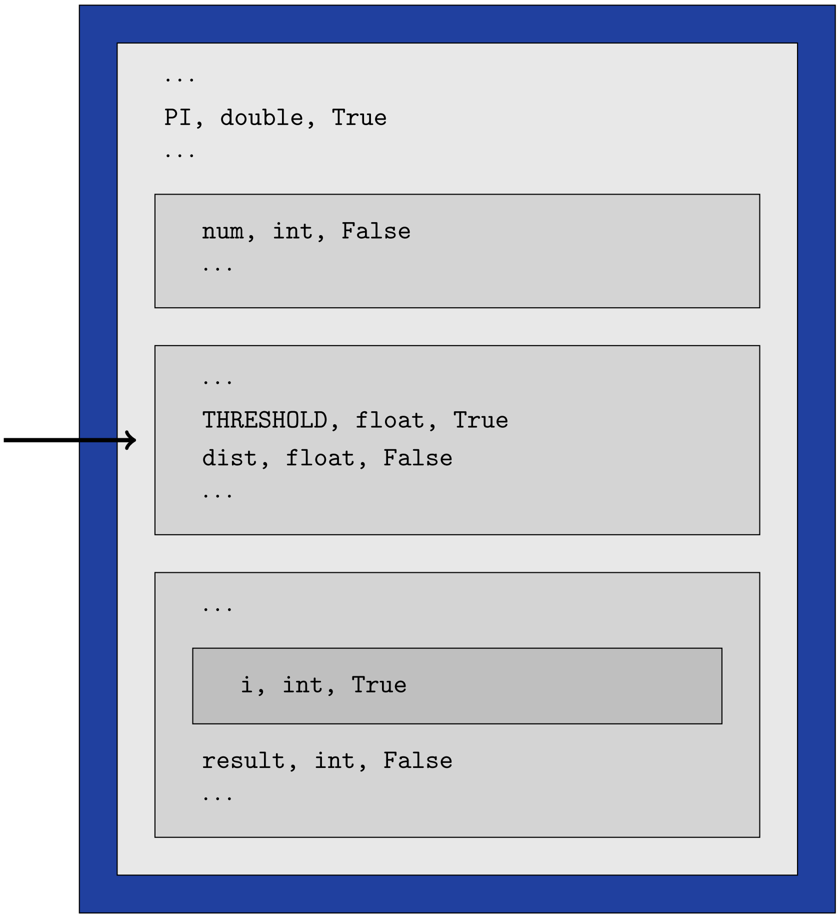

Figure 4: Illustration of the hierarchy of symbol table classes.

4 Intermediate code

We uses the three-address code format for the our internal intermediate code, i.e., our intermediate code is the form of

```
x = y op z.
```

The detailed specification of intermediate code for each statement does not defined yet. However, we try to specifying it in the next week.

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.