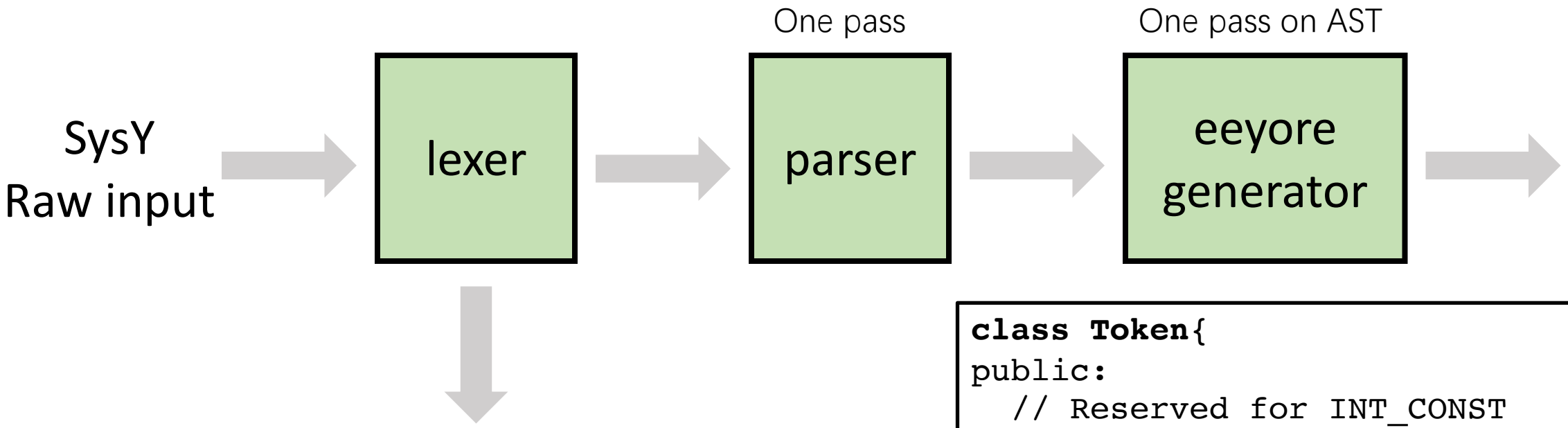
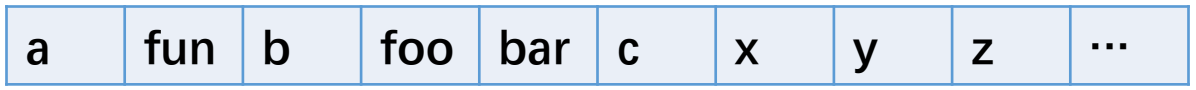
Part 1 Token: A simple and uniform class



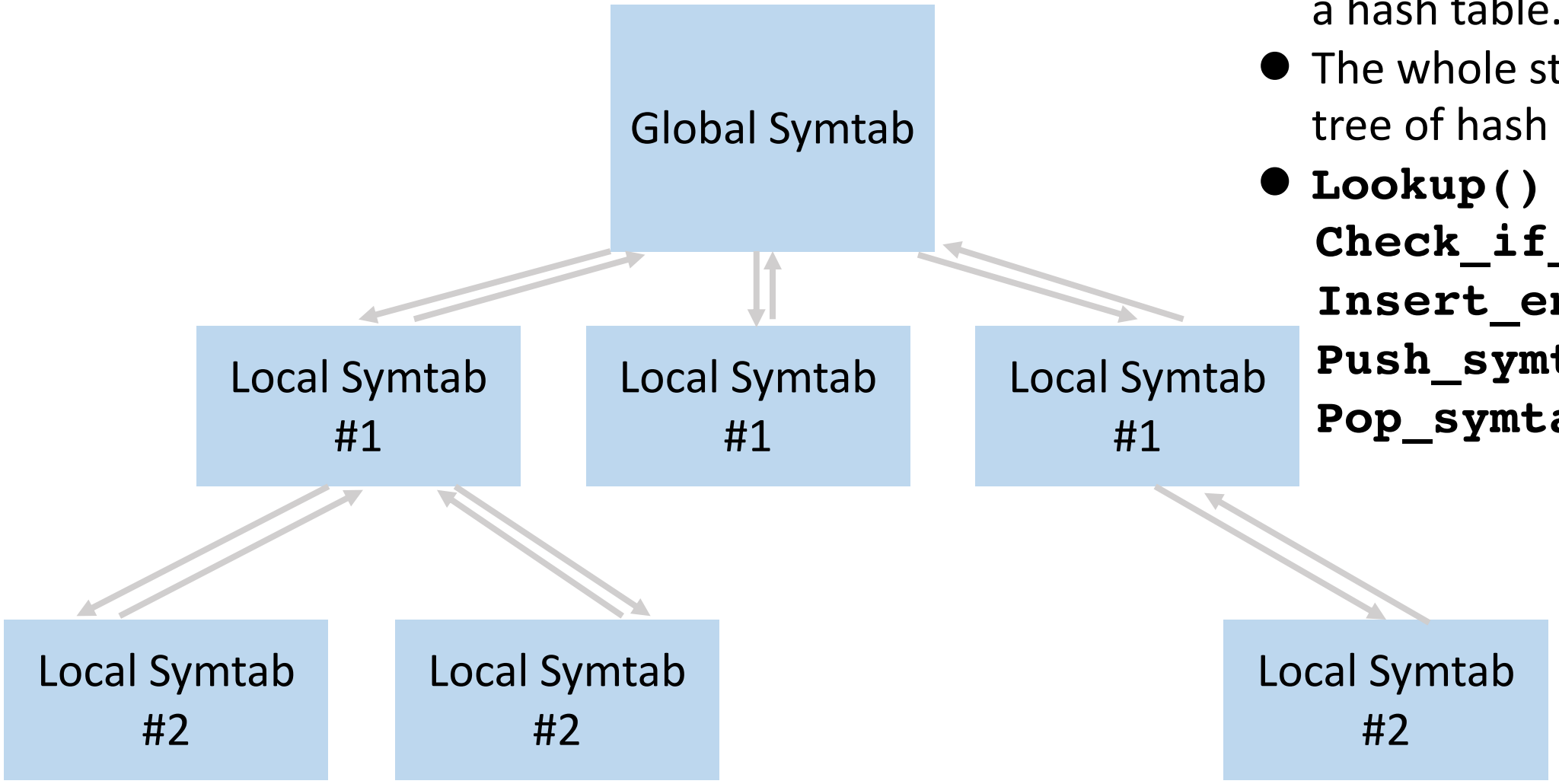


String Table

Purpose: To eliminate the need of moving or duplicating literal strings. Always move or copy pointers that are into string table.

```
class Token{
public:
  // Reserved for INT_CONST
  int val;
  // Reserved for IDENT
  STRptr strptr;
  // token type, defined in header
file generated by yacc. For later
usage.
  int type;
  // Position info. For error report.
  int lno, bgn, end
};
```

Part 2 Symbol Table (Maintained by parser)



- Each node stores the vars and funcs declared within that scope, and is implemented by a hash table.
- The whole structure is thus a tree of hash table.
- **`Lookup()`**
  `Check_if_declared()`
  `Insert_entry()`
  `Push_symtab()`
  `Pop_symtab()`

Part 2 Symbol Table (Maintained by parser)

For convience, a field within a symtab entry points back to AST node that declares it.
(We store the position info in AST node to facilitate error report.)

VarSYM:

Store dimension info. (It can always be evaluated in compiling time.)

Is it constant?

A **uniform** representation of both array and pointer: (Always store the <…> form)

e.g.
```
int a[2][3][5];   ->   <1, 5, 15, 30>
int a[][3][5];    ->   <1, 5, 15, 0>
int a;            ->   <1>
int a[1];         ->   <1, 1>
int a[];          ->   <1, 0>
```

Possibly there are initial values, we store it in the corresponding AST node.

Part 2 Symbol Table (Maintained by parser)

For convience, a field within a symtab entry points back to AST node that declares it. (We store the position info in AST node to facilitate error report.)

FunSYM:

Ptrs to all the symtab entry of its formal parameters.

Does it return void or int?

Note that FunSYM has to be in the symtab of **the upper laye**r, while its formal params has to be in the symtab of layer **underneath** it.

-> To achieve this **in one pass**, you may want to **tweak the EBNF** defining SysY grammar.

-> btw. **the counter of scoping** can be similarly implemented in the parser in just 4-6 lines in quite a simple manner. ***How to do this?***

-> You should be aware that there are **two kinds of scopes.** One is the normal { } that encircles a block (not a list initializer such as int a[3] = ‘{‘ 0, 1, 2 ’}’;); the other is (} of a function definition. (e.g. void fun ‘(‘ int a) { return; ‘}’). This way, we allow self-recursive function and constrain the use of parameters strictly within that function.

Part 3 AST (builded up by parser, and used in IR generation)

We would want AST have these in common (which naturally give rises to a base AST):

- Position info for error report (should be at least a 4-tuple **<bgnlno, bgnpos, endlno, endpos>**)
- Type info suggesting its full type (sometimes is useful in polymorphism)

Two tool classes are introduced for our convenience, for which I coined the name “the scaffolding class”. They work in a way that resembles scaffolding in the erection of a building: They help to ***structurize*** the irregular component, ***rearrange*** them in order, provide a ***structural*** representation, and finally they ***vanish***, leaving the grand edifice alone.

- VecAST whose member is simply vector<ASTptr>;
  Quite often we have to collect a list of sub AST-tree. E.g. Formal/Real parameter list.
- ListTree whose kernel function is to check the consistency of a list initializer and flat them down linearly.
  e.g. int a[3][2] = {{}, a, b, {c}}; -> {(2, a), (3, b), (4, c)};
  This can be achieved in light of ***LISP***.

Part 3 AST (builded up by parser, and used in IR generation)

Other AST:

- BinaryAST Store a binary op, e.g. '-', '&&', '<='. The mixture of arithmetic op, rel op and logic op ***will not*** incur clutter in Eeyore generator, as we shall see.
- UnaryAST Store a unary op, e.g. '!', '-'
- FunCallAST It stores the ptr into symtab entry of the calling function, and a vector of real params (since each param is an AST).
- LvalAST A ref of variable. It stores the ptr into symtab entry of the referencing variable. Also, it stores the indices. (In case of 'int a;', the indices will just be empty.) Each indices is also an AST.
- BlockAST It stores a vector of AST (within a block)
- DeclAST Definition of a variable. It stores a ptr into symtab, and the initial value (As we have seen, it is a map). Note that dimension info can be fetched from the symtab entry.
- FunDefAST Definition of a function. It stores a ptr into symtab, and a vector of ASTptr into the DeclAST block of its formal params. Note that symtab entry of its fparams can be fetched easily. Also, it stores an ASTptr to the body of the function.

Part 3 AST (builded up by parser, and used in IR generation)

Other AST:

- AssignAST Marks an assignment. It has ptrs to a LvalAST and Exp respectively.

It is important to notice that Exp can end up on BinaryAST / UnaryAST / FunCallAST / LvalAST only.

- IfAST Marks an if-statement. Cond, then-statement or an optional else-statement.
- WhileAST Similarly.
- BreakAST A break statement. It has a ptr into the WhileAST that it breaks from.
- ContAST A continue statement. It has a ptr into the WhileAST that it is within.
- RetAST A return statement. It has an optional ptr into the Exp tree it is about to return.

Aside: Error Report

Both token and AST represent a segment of source code. In order to generate user-friendly error report, one needs to display exactly the position of occurrence as well as possible error type. But may choose whether to supply remedy. (Here we may not)

Token must be on the same line, thus a 3-tuple **<lno, bgn, end>** suffices.
Since an AST node can be on different lines, a 4-tuple **<bgnlno, bgnpos, endlno, endpos>** will suffice.

Also, many errors occur at a semantic level. Ideally we would love to use the links between AST, and symbol table.

```
void fun(int a){
  return
  ^ a * a
  + a;
}    ^
// ...
```

An error could stride consecutive several lines

```
// ...
void fun(int a);
// ...   ^~~~
  const int a;
  fun(a); //
// ...^
```

A semantic error, where we want to highlight both the ref and def