Syntax + Semantics = Language

While the grammar for a language enforces syntactic constraints on accepted strings, some language issues are often postponed until after parsing.

For example, some language definitions contain rules that cannot be enforced by any context-free mechanism.

The most common examples involve some form of *type-checking*. Recall our expression grammar, a form of which appears in most programming language grammars. While the grammar allows an expression such as

$$a + b$$

most languages contain rules that restrict the types of a and b. For example, addition does not make sense if a is a character string and b is an array.

A grammar that accommodates type information would involve some *context*, and such grammars are difficult to design and expensive to process. Viable approaches to this problem involve some form of semantic processing, performed during or shortly after parsing:

Attribute grammars
: **specify equations whose resolution essentially performs type checking.**

Symbol tables are the most common solution. Type information is entered when identifiers are declared, so that expression types can be subsequently checked.

There is still the issue of whether type checking occurs in the same *pass* over the input as syntactic checking. Some languages forbid the kinds of "forward" declarations that would require extra passes for type checking.

Semantic processing

Also, there are often language constraints that are difficult or unwieldy to enforce syntactically.

For example, the ANSI C grammar essentially has a set of rules:

Declaration	$\rightarrow$	**Qualifiers id**
Qualifiers	$\rightarrow$	**Qualifiers Qualifier**
	$\mid$	**Qualifier**
Qualifier	$\rightarrow$	**int**
	$\mid$	**float**
	$\mid$	**static**
	$\mid$	**extern**
	$\mid$	$\vdots$

While this grammar allows strings like

```
static int x
```

the grammar also admits strings such as

```
static int extern float x
```

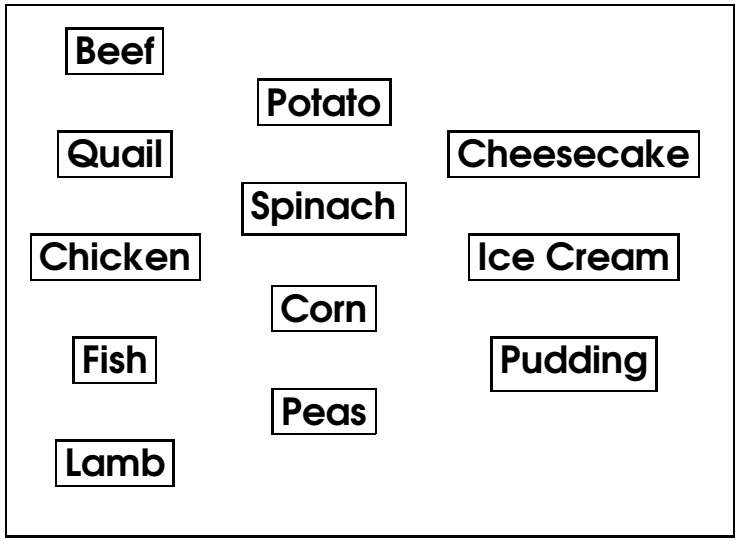
The language actually offers three kinds of type qualifiers. At most one from each category is allowed for any identifier.

The grammar could be transformed to enforce the kind and number of qualifiers that are allowed, but this would increase the size of the grammar.

Another example would be the *evaluation* of an expression. If we restricted the size of its terms, each expression could be syntactically evaluated by a huge grammar. Taken further, any programming language can be processed by a finite-state machine if the program size is bounded.

Ultimately, issues of taste and efficiently dictate how and where language issues are addressed.

Ordering from a Chinese menu



The rules for a "correctly" placed order are:

1. **At most one item may be selected from *any* column.**
2. **Some columns may be skipped.**
3. **At least one item must be chosen.**
4. **The items can be arbitrarily ordered.**

```
Order   → Choices
Choices → Choice Choices
        | Choice
Choice  → ColA
        | ColB
        | ColC
ColA    → BEEF
        | CHICKEN
        | QUAIL
        | FISH
        | LAMB
ColB    → POTATO
        | SPINACH
        | CORN
        | PEAS
ColC    → CHEESECAKE
        | PUDDING
        | ICECREAM
```

The assignment is to rewrite the grammar to enforce the rules. This is exactly what's needed to enforce C's rules for declarations.

Solution

Order	$\rightarrow$	**Choices**
Choices	$\rightarrow$	**ColA**
	$\mid$	**ColB**
	$\mid$	**ColC**
	$\mid$	**ColA ColB**
	$\mid$	**ColB ColA**
	$\mid$	**ColA ColC**
	$\mid$	**ColC ColA**
	$\mid$	**ColB ColC**
	$\mid$	**ColC ColB**
	$\mid$	**ColA ColB ColC**
	$\mid$	**ColA ColC ColB**
	$\mid$	**ColB ColA ColC**
	$\mid$	**ColB ColC ColA**
	$\mid$	**ColC ColA ColB**
	$\mid$	**ColC ColB ColA**

ColA	$\rightarrow$	**BEEF**
	$\mid$	**CHICKEN**
	$\mid$	**QUAIL**
	$\mid$	**FISH**
	$\mid$	**LAMB**
ColB	$\rightarrow$	**POTATO**
	$\mid$	**SPINACH**
	$\mid$	**CORN**
	$\mid$	**PEAS**
ColC	$\rightarrow$	**CHEESECAKE**
	$\mid$	**ICECREAM**
	$\mid$	**PUDDING**

While some factoring of this grammar is possible, this example illustrates the tradeoff between grammar size and specificity of the parse.

Symbol tables

The symbol table tracks symbols and their types, where type information could be any property of a symbol relevant to subsequent activity in the compiler.

```
static char *a[5];
```

is an array of 5 pointers to characters.

Such information typically includes

- **the basic type of a variable (ptr, int, char, float, struct, etc.);**
- **structure layout, pointer specifics, array information;**
- **initialization values;**
- **scope information.**

I provide the following symbol table access functions:

IncrNestLevel(): increase the nest level by one.

DecrNestLevel(): decrease the nest level by one.

EnterSymbol(M,name): enters the string **name** as a symbol of type **M** at the current nest level.

RetrieveSymbol(name): returns a pointer to the currently active declaration of **name**. If **name** is not active, an error message is produced and the parse is aborted.

ExistsSymbol(name): operates like **RetrieveSymbol()**, but instead of aborting, a NULL pointer is returned if **name** isn't active.

I provide extra credit for those who implement their own, hash-based symbol table manager.

Symbol tables

Essential information

1. **Names;**
2. **Scope information;**
3. **Type information;**
4. **Storage specifics.**

Issues

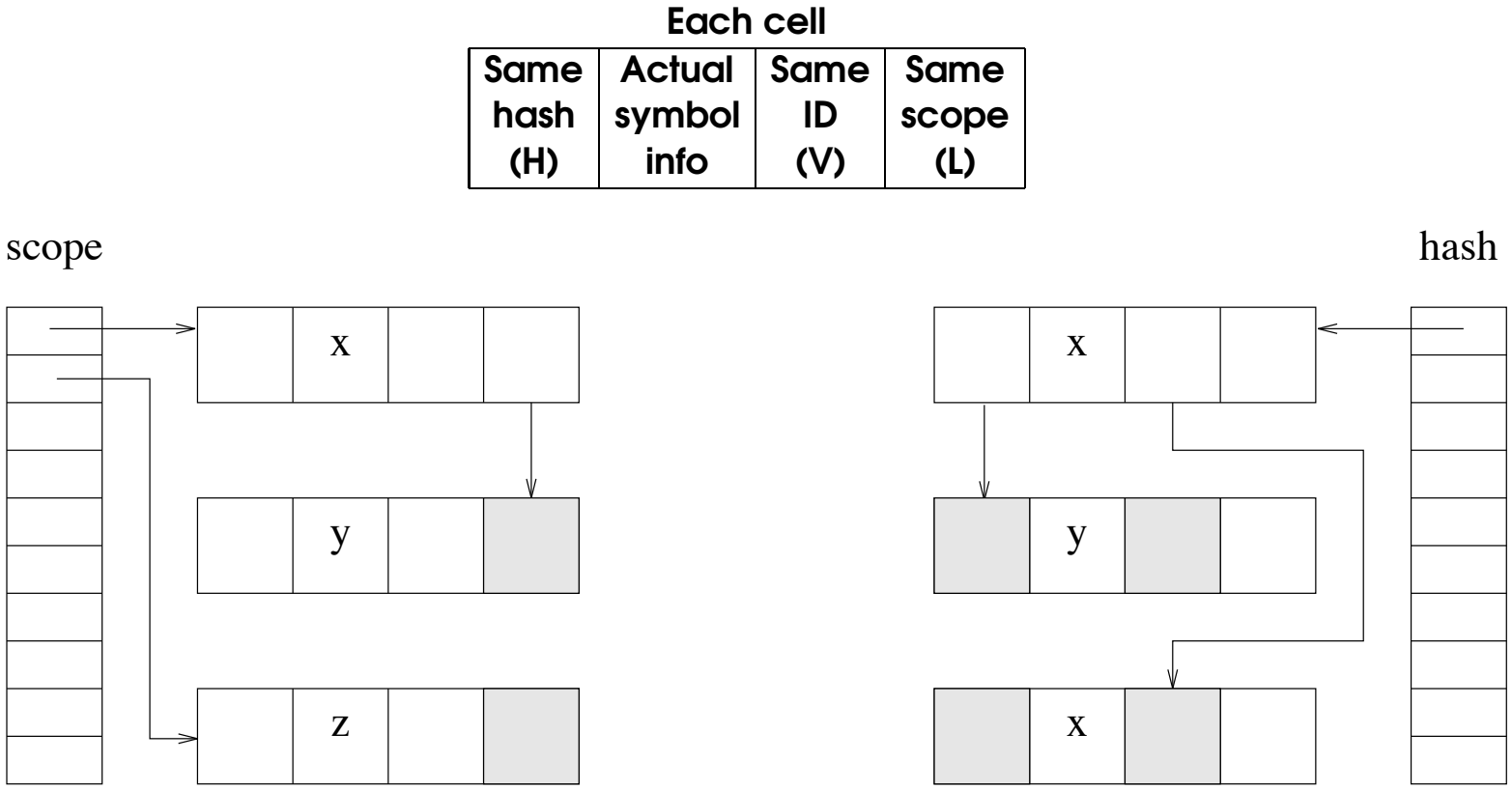
1. **Programs typically contain a mix of very long and very short names (`i` *vs.* `WindowMaxAccelScreenMouse()`).**
2. **Type checking and code generation do not require access to all scopes at all times. Typically, access is required only to the current scope and its outer scopes. Even then, programs use the current and outermost scopes most frequently.**

There are two popular methods of establishing symbol tables:

1. Make a separate pass over the program to create the symbol table;
2. Build the symbol table as you parse.

Given that one typically creates an abstract syntax tree anyway, it seems wise to defer symbol table creation to a separate pass. On the other hand, restructuring the grammar to simplify symbol table creation is a good exercise, and it is necessary for a one-pass compiler.

Symbol table organization



The above scheme implements a stack for each variable v, where top-of-stack is the currently active instance of v. Let $f(v)$ be the hash index for variable v:

Entering a scope: each variable v is pushed onto the stack headed by its (chained) hash index $f(v)$.

Leaving a scope k: each variable linked from scope k is popped off its stack.

Lookup: use $f(v)$, with chaining via H, to locate the named variable.