Reading Assignment

Fischer & LeBlanc

Chapter 7

Chapter 8

Syntax Directed Translation

- In one pass compilers the most common organization is for the syntax analysis phase (parser) to direct the entire compilation
- The parser calls the scanner as require to get tokens.
  At various points during the parse the parser calls the semantic analysis and code generation phases to perform various actions.
- The semantic and code generation actions are often described using a grammar for the programming language.
- In multi-pass compilers, the parser builds some data structure that represents the program being compiled.
  Semantic analysis and code generation are implemented as algorithms that traverse this data structure.
- If the language being processed is context sensitive (e.g. C) then some semantic analysis must be done during parsing to allow parsing to proceed.

Semantic Analysis

- Validation of *non-syntactic* language constraints
  Static semantic analysis - during compilation
  Dynamic semantic analysis - run time checks
- Semantic analysis
  - Visibility and Accessibility Analysis
  - Type Checking
  - Proper Usage Analysis
  - Range and Value Analysis
  - Range and Value Propagation
- Semantic analysis usually builds the compilers symbol table as it processes the program.

Programming Language Influences

- Definition of the programming language being compiled can have a major influence on the way semantic analysis is implemented.
- Some issues:
  - Languages that do not require *declaration before use* (e.g. PL/I, C) will usually require a separate semantic analysis pass (or a lot of kludgery).
  - Languages with a context sensitive syntax (e.g. C, Fortran) will require feedback from the semantic analysis pass to the parser or scanner.
  - Language with a weak or non-existent declaration structure (.e.g. Lisp, Icon, APL, Prolog) require that most semantic analysis be done dynamically.
  - Object-Oriented languages (e.g. Smalltalk, C++) that allow dynamic object binding may require extensive run time checking.
  - The presence of dynamically sized objects in a language (e.g. arrays whose bounds are determined at run time) may require more run time checking.
  - Compilers for *student* languages should do very thorough static and dynamic semantic analysis.

Scopes and Declarations

- Assume that a program consists of some number of nested *scopes* of declarations (e.g. main program, **begin** - **end** blocks, functions and procedures).
- Identifiers are declared in one or more scopes.
- The *scope rule* for the programming language determines the visibility of of symbols declared in one scope in other scopes.
- The most common scope rule is the *Algol-60 scope rule* which allows identifiers declared in a scope to be automatically visible in all contained (properly nested) scopes.
- Languages with modules, objects, packages or classes often have different visibility rules for identifier declared within those constructs.
  (e.g. C++ classes allow declared identifiers to be **public**, **protected**, or **private**)

Visibility and Accessibility

- Visibility analysis determines whether a given reference to an identifier at some point it the program is legal according to the scope rules of the language.
- The perform visibility analysis, the compiler must keep track of declared symbols behaves (logically at least) in a way that is consistent with the scope rules of the language.
- Semantic analyzer must also track the scope structure of the program as it is being processed.
- Accessibility analysis determines whether a visible identifier can be accessed in a given way at some point in a program. Examples
  bind in Turing restricts access to bound to identifiers
  const in C prevents variables from being assigned to
  or having their address assigned to non-const pointers.
  anti-aliasing detection in original Turing

Type Equivalence, Compatibility, Suitability

- Every language definition includes rules about when objects of different types can be used together in various constructs.
- Many languages have several rules concerning types:
  - *Type Equivalence* Conditions under which objects of two different types are considered to be *equivalent*.
    Equivalence is usually required for reference parameters and pointer objects.
  - *Type Compatibility* Conditions under which objects of two different types are considered to be *compatible*.
    Compatibility is usually required for assignments and value parameters.
  - *Type suitability* Conditions under which objects of two different types are considered *suitable* to be used together.
    Suitability is usually required for expressions.
- The two most widely used Type Equivalence Rules are *structural equivalence* and *name equivalence*

Type Equivalence Rules

- Define: *Name Type Equivalence*
  - Two types are name equivalent iff they ultimately derive from a common definition.
  - *ultimately derive* allows type renaming, e.g. **type** S : T
  - In implementation terms, two named types are equivalent if they refer to the same type table entry.
- Define: *Structural Type Equivalence*
  - Type types are structurally equivalent if their definitions have the same structure and corresponding values are equal.
  - Structural equivalence is isomorphism for types.
  - In implementation terms a parallel walk of two type trees is required to establish structural equivalence.

Usage Analysis

- Appropriate use of constants, variables, types, procedures, functions.
- Enforcing language and implementation limits.
- Enforcing anti-aliasing and other safety constraints.
- Detection of potential arithmetic faults.
- Detection of potential run-time faults.
  Emitting code for dynamic semantic analysis.

Value and Range Analysis

- Range checking of array subscripts.
- Checking assignment to subrange variables.
- Enforce language/implementation limits.
- Range propagation to minimize run time checking.
- Range Propagation
  - Each constant and variable has an inherent range.
  - For each arithmetic operation on range variables there is an expression that gives the range of the result.
  - Propagate ranges through expressions to determine range of final result.
  - Use propagated range information to minimize run time checking and to detect potential run time arithmetic faults (e.g. divide by zero, overflow).

Semantic Analysis Example

	X	:=	Y		
Visibility	declared(X)?		declared(Y)?		
	visible(X)?		visible(Y)?		
Accessibility	access(X)?		access(Y)?		
	write(X)		read(Y)?		
Usage	variable(X)?		variable(Y)?	constant(Y)?	function(Y)?
Type	type(X)?		type(Y)?	type(Y)?	type(Y)?
	scalar(X)?		scalar(Y)?	scalar(Y)?	scalar(Y)?
Usage	assign(X)?				parameters(Y)?
Type		compat(X,Y)?			
Value/Range	range(X)?		range(Y)?	value(Y)?	range(Y)?
		inRange(X,Y)?			

Usage Analysis Example

Assume

type i : 1 .. 23
var x : **integer**
y : i
a : **array** [1 .. 10] **of char**
function f (p : **integer**) : **integer**

Statement	Error
i := 17	Assignment to a type
f(3) := x	Assignment to a function value
x := a	Assign array to scalar variable
x[y] := 31	Subscript on a scalar variable
x := y.q	Field selection on a scalar variable
a := f(a[1], y)	Assign integer to char array
	Wrong number and type of parameters to f

Range Propagation Example

Assume

var i,j : 1 .. 100
m,n : **integer**
k : -50 .. 40

Statement	RHS Range	Checks
i := i + 1	2 .. 101	check i <= 99
i := n + 1	minInt+1 .. maxInt+1	check n < maxInt
		check 1 <= n+1 <= 100
n := i * j	1 .. 10000	
m := k * j - 10*i + 40	k*j -5000 .. 4000	10*i 10 .. 1000
	k*j-10*i -6000 .. 3990	40 40 .. 40
	-5960 .. 4030	

Arithmetic Fault Detection Example

Assume

var i,j,k : **integer**
m : 0 .. 100
n : -100 .. 100
/* Assume 16-bit integers */
/* minInt = -32768 maxInt = 32767 */

Expression	Potential fault
i := i / n	divide by zero
k := k + 1	overflow
j := - i	overflow
i := 33 * m * n	underflow, overflow

Managing Semantic Analysis Data

- The semantic analysis phase will typically require some data structures to keep track of semantic information as it processes the program.
- Due to the *embedding* that occurs in most programming languages (e.g. statements within statement, expressions within expression) stack-like data structures are usually required.
- Toy, prototype and student compilers often use fixed sized stacks for various internal data structures. Production compilers often allocate data structures dynamically on demand and use linked lists instead of stacks.
- In single pass compilers, a commonly used approach is to use several stacks that operate *in parallel* with the parse stack.
  For each item on the parse stack, the corresponding parallel stack entries might hold type, symbol, value, address or other information.
  Use right recursive definitions to capture an entire construct (e.g a case statement) in the parse stack.

Semantic Data Examples

- Stacks parallel to parse stack

Parse	Type	Symbol	ValType	Value
then				
0	integer		intConst	0
=	relOp			
identifier	integer	syt(I)	VarAdr	200(4)
if			CodeAdr	2420
statements			CodeAdr	1456

- Right recursive definition

caseStmt → casehead caseList caseEnd
caseList → caseClause
→ caseClause caseList
caseClause → **case** label : statementList

Compiler Tables & Dictionaries

- Most compilers make extensive use of tables to capture information about the program being compiled.
- The *Symbol Table* contains information about declared things (usually identifiers).
- Compilers for modern languages often use a separate *Type Table* to record information about built in and user declared types.
- Table management is often a major performance issue in the design of a compiler
- Table usage
  - Declaration processing - add entries for identifiers being declared
  - Semantic analysis - look up identifiers to determine their attributes.
  - Code generation - look up identifiers to determine their attributes.
  - Table lookup must follow the languages scope rules.

Generic Symbol Table Entries

- A typical symbol table entry might contain:
  - Name of the item
  - Kind of item, e.g. constant, variable, type, procedure, function, etc.
  - Type of the item (index into type table)
  - Attributes or properties associated with the item (usually language specific)
  - Size of the item (or derive from type)
  - Run time address for the item, e.g. base register and offset.
  - Value for the item, e.g. value of constants, initial value for variables (might be index into a table of constants)
  - Links to related symbols, e.g. parameter lists, enumerated constants, fields in a struct or union.
- Usually one symbol table entry per declared symbol, low hundreds for student, toy compilers, in the thousands for production compilers.

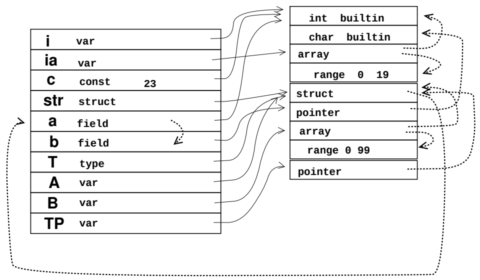
Generic Type Table Entry

- A typical type table entry might contain:
  - Name of the item (often omitted)
  - Kind of type e.g. typedef, enum, scalar, array, struct, union, etc.
  - Attributes or properties associated with the type.
  - Size and memory alignment for objects of this type.
  - Actual definition of the type, usually involves links to embedded components.
  - Links to related definitions.
  - In most modern languages, the type table is a DAG.

Symbol and Type Table Example

```
typedef
  struct str {
      int a ;
      char * b ;
  } T ;
```

```
int i, ia[20] ;
const int c = 23 ;
T A, B[ 100 ], * TP ;
```



Handling Related Entries

- There are cases where the compiler needs to keep track of related symbol and/or type table entries.
  Examples: fields in a struct, the parameters of a procedure or function, the names in an enumerated type.
- One approach is to assign consecutive table entries to related items so that the index of the first item and an offset is sufficient to locate any item.
  This approach may be a considerable pain to implement if embedding in the language requires that table entries be build from the inside out, e.g. struct definition containing an embedded struct definition.
- Another approach is to use explicit links (pointers or array indices) in the table structure. This costs table space, but is easier to build during declaration processing.

Handling Values

- The compiler needs to be able to represent in its tables any kind of *value*, i.e. constant, that could occur in a program.
- To conserve table space, some form of union data structure is often used:

```
struct constantDesc {
     short  constantKind ;
     union  {
            int    intValue ;
            float  floatValue ;
            char   charValue ;
            char  *stringValue ;
            void  *bigValue ;
     } constantValue ;
}
```

- Large constant values, e.g. initialization for arrays or structs often require special storage.

Scope Rules and Tables

- In most programming languages a program is partitioned into some (possibly overlapping) scopes of declaration.
- The scope rule for a language (see Slide 138) specifies the algorithm that must be used to search compiler tables.
- To implement most scope rules, each scope of declaration is *logically* a separate set of symbol table entries.
  It is often useful to handle major scopes (main program, modules, procedures and functions) differently from minor scopes (bodies of **if** and **for** statements).
- For the most common (Algol-60) scope rule, scopes are properly nested and the symbol table behaves in a strictly stack-like fashion.
- Most compilers distinguish symbol *visibility* from symbol *storage*. The scope rule determines which symbols are visible at any point in the program.
  Symbols may remain in the symbol table even if they are not visible (e.g. entries for the parameters of a function).

Symbol Table Operations

The following operations are typical of a symbol table module

- Create Symbol Table
- Enter new scope
- Exit Scope
- Lookup symbol in current scope
- Lookup symbol using scope rule
- Enter new symbol in current scope
- Enter new symbol in designated scope
- Delete symbol from table
- For all fields in the symbol table
  - Set value of field for symbol
  - Get value of field for symbol
  - Modify value of field for symbol
- Language specific operations

Table Search Strategies

- Most symbol table operations are table lookup. A typical symbol gets entered once when it is declared, looked up at least twice (semantic analysis and code generation) every time it is used. Deletion is usually done wholesale by discarding all the entries for a scope.
- For searching small scopes or for toy and student compilers *linear search* is a good alternative. Search top of symbol stack to bottom automatically implements Algol-60 scope rule.
- Use binary search for static tables, i.e. lists of reserved words, lists of builtin functions.
- Hash tables are the preferred search method for production compilers. O(1) search if the table is large enough.
- Indexing search is a good choice if the scanner already maps identifiers to small integers.

Table Search Strategies

- Indexing Search
  - Scanner maps all identifiers into small integers.
    Cost one lookup per occurrence. Call this integer the symbol index.
  - A vector (symbol vector) indexed by symbol index points at the most recent symbol table entry for the identifier.
  - On declaration, save current index for identifier in new symbol table entry, point symbol vector entry at new symbol table entry.
  - On scope exit, restore previous symbol vector entry for each symbol in the scope.
  - Symbol lookup through the symbol vector is always O(1).
- Builtin symbols (library routines) are often handled by wrapping a meta-scope around the entire program and building symbol table entries for the builtin symbols there. This permits automatic redeclaration of builtins.

Hash Tables

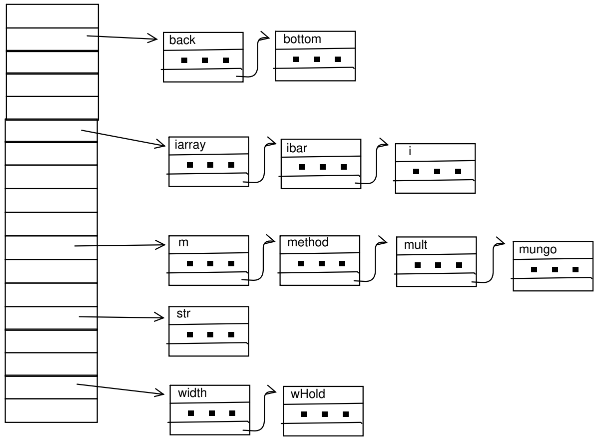
- Preferred Organization for Symbol Tables
- Design issues
  - Hash function
  - Collision resolution
  - Interaction with scopes
  - Efficiency of search, insertion, deletion
- Hash function
  - Should be *simple* to compute and provide good dispersion.
  - Typical hash functions are some arithmetic combination of letters from the identifier and/or the length of the identifier.
  - Spending a *lot* of time optimizing the hash function probably isn't worthwhile, simple functions like product of first and last letters , or sum (xor) of letters is good enough.
  - Pro Compiler: have scanner compute hash function *once* for each identifier and package that value with the internal representation of identifiers.

Hash Tables (continued)

- Hash function is a mapping from identifiers to hash table indices. Since this (must be) a many-to-one mapping, collisions *will* occur when two different identifiers map to the same index.
- The preferred method for dealing with collisions is resolution by chaining since it supports symbol table scope operations well.
- Each hash table entry is the head of a linked list of identifier nodes. The hash function is used once to locate one table entry, then the linked list is searched serially to find a given identifier.
- Advantages of using chaining to resolve collisions are
  - It allows arbitrarily large symbol tables since table storage is allocated incrementally for each identifier.
  - It facilitates removing identifiers from the table at the end of a scope.
- Indexing search (Slide 159) is a form of hashing with optimized collision resolution by chaining that uses a one-to-one hash function.

Hash Table Example

Hash Table with collision resolution by chaining.



Fisher/Lablanc Figure 8.2

Minimizing Table Space

- The symbol table for most compilers is designed to hold a *large* number of entries. Thousands of symbol table entries would not be unusual in compiling a large program.
- It is therefore good design practice to try and keep symbol table entries as small as possible. One approach to minimizing the size of symbol table entries is to use indirect reference via pointers for any large things that would otherwise by in the symbol table.
- Many compilers use a separate table for storing the actual names of identifiers since otherwise a lot of (potentially wasted) space would be used in each symbol table entry. This optimization is particularly useful if the language allows very long identifiers, e.g. 32 characters in PL/I, 50 characters in Turing
- The semantic analysis and code generation operations performed by a compiler generally *do not need* the actual name of each identifier as long as there is some index value that maps one-to-one to identifier names.

Managing Scope Information

- There are several alternatives for designing symbol tables to implement scoping in programming languages.
- Separate symbol table per (major) scope.
  Allocate on scope entry, free on scope exit. Compiler keeps stack of pointers to currently active scopes.
  This table could be a stack or separately allocated chunks of memory.
- Global symbol table.
  Allocate at start of compilation. Add symbols as required.
  At the end of a scope (logically) delete all identifiers declared in the scope from the symbol table.
  Using hash table with chained entries, insert new symbols at head of chain, delete symbols from head of each chain until symbol from previous scope is uncovered or chain is empty.

Symbol table design issues

- *Statements modifying visibility.* Constructs like **with** in Pascal and **bind** in Turing change the visibility of symbols in the middle of a scope. These changes can be supported by copying symbol table entries or by using flags in the symbol table to render symbols invisible to the search process.
- *Micro Scopes* Many languages allow very small scopes (e.g. the bodies of **if** and **for** statements in Turing)
  It is usually more efficient to store symbols for micro scopes in the symbol table of the enclosing major scope (procedure or function) and control their visibility with symbol table flags.
- *Record fields.* The names of record fields should only be visible inside the record. Such field names should be stored in a way that makes them invisible to normal search.
  Processes records inside out to deal with nested record declarations.

Symbol table design issues - continued

- *Retained names/entries.* For some types of scopes, names and/or symbol table entries must be retained in the symbol table after the scope has been processed. Examples include the names exported by a module and the formal parameters of a procedure or function.
- *Information Hiding.* Constructs like **opaque** in Turing allow the programmer to hide information about the actual data representation of a type while still allowing the type to be used to create variables.
- *Importation.* If a module or similar construct imports some names from its environment, these names need to be made available in the modules local symbol table.

Symbol table design issues - continued

- *Implicit Declarations* In some languages the *occurrence* of an item constitutes a declaration for the item. Examples: statement labels, previously undeclared functions in C and Fortran, **for** loop indices in Turing.
  To handle implicit declaration the compiler must be prepared to create new symbol table entries for any undeclared identifier that it encounters.
  There may be scope related interactions for some kinds of symbols .e.g.
  for i := 1 **..** i+3 **...** **end for** in Turing.
- *Name Overloading* Some languages (Ada, PL/I, C++) allow multiple definitions for the *same* name in cases where there is some language specific rule for disambiguating uses of the name.
  For example, Ada allows multiple definitions for the same function as long as the definitions have distinct formal parameter lists.

Symbol table design issues - continued

- *forward references.* Many languages allow implicit or explicit pre-declaration of names.
  Examples: label appearing in a **go to** statement, the explicit **forward** declaration in Turing, the use of function headers to pre-declare functions in C.
  This usage can be implemented by creating symbol/type table entries that are marked with an *IamForward* flag. If the corresponding real declaration is later encountered, it is validated against the forward declaration and the flag is turned off. At the end of a scope the symbol table entries for the scope need to be checked for unsatisfied forward declarations.
- *Supporting separate compilation.* Some compilers support separate compilation at the file or module level by saving the symbol table for names exported from the file or module on disk in a compressed form. Any compilation that needs this symbol information can efficiently read it from disk.
  In contrast, almost all C compilers process `#include` directives by reading and processing a source file.

Pascal Tables Example - Base Definitions

```
#define maxString    500     /* size of string constant table */
#define maxSymbols   600     /*size of symbol table */
#define maxTypes     400    /* size of type table */
#define maxName        8    /* max length of names */
#define nullIndex      0    /* null table index */

/* Kinds of Symbols */
enum symKind { unknownSym, constSym, typeSym, varSym, procSym,
          funcSym, labelSym } ;
/* Kinds of Types  */
enum typeKind {unknownType, intType, charType, realType, boolType,
               enumType, stringType, subrangeType, pointerType,
               forwardType, arrayType, recordType, setType,
               fileType, vrField } ;
typedef  short  symIndex ;       /* symbol table index */
typedef  short  typeIndex ;      /* type table index   */
typedef  short  stringIndex ;    /* string table index */
typedef  char [ maxName ] name ; /* symbol name */
```

Pascal Tables Example - Descriptors

```
typedef struct    mAddrStruct{  /* machine addresses */
             unsigned base  ;
             integer offset ;
          } mAddress ;
typedef struct lAddrStruct{    /* logical addresses */
            unsigned lexLevel ;
            unsigned ordNumber  ;
       }  lAddress ;
typedef struct constStruct {  /* description on constant's values */
       short cType ;   /* type of constant */
       union {
            integer       iValue ;       /* integer */
            char          chValue ;      /* char    */
            float         rValue ;       /* real */
            unsigned char bValue ;       /* Boolean */
            integer       eValue ;       /* enum */
            stringIndex   sValue ;    /* string */
       } constDesc ;
typedef struct pfdStruct {   /* procedure/function descriptor */
       mAddress  pfAddress ;  /* code address */
       unsigned  pfCount   ; /* parameter count */
       symIndex  pfParml   ; /* syt 1st parameter */
     } pfDesc ;
```


Pascal Example - Type Table Entry

```
/* definition of type table entry */
typedef struct typeStruct {
       unsigned tySize ;    /* size of this type */
       /* fields that depend on the kind of type */
       typeKind tyKind ;    /* type kind for this type */
       union {      /* information depending on tyKind */
      struct {                    /* enumerated type */
            short eCount ;        /* number of values */
            symIndex eValue1 ;    /* ptr to 1st value constant */
            } enumInfo ;

      struct {                    /* sub range type */
          typeIndex srBase  ; /* ptr to base type */
          constDesc srFirst, srLast  ; /* bounds */
         ) subRangeInfo  ;
      unsigned strMaxLength  ;    /* string */

      struct {                  /* array */
           boolean aPacked  ;    /* packed array */
           unsigned  aCount  ;   /* element count */
           typeIndex aIndex  ;  /* ptr to index type */
           typeIndex aComponent ;   /* ptr to component type */
         ) arrayInfo ;
```


Pascal Example - Type Table Entry (continued)

```
 struct {      /*  record type */
      boolean   rPacked ;    /* packed record */
      unsigned  rCount  ;    /* number of record fields */
      symIndex  rField1 ;    /* ptr 1st field */
      boolean hasVariant ;   /* contains variant record */
      symIndex  vrTag   ;    /* var record tag identifier */
      typeIndex vrType  ;    /* variant record type entry */
      typeIndex vrVariant1 ; /* ptr to first variant      */
    }  recdInfo ;

struct {    /* pseudo type entry for variant record fields */
      constDesc vrLabel  ;       /* value of label */
      typeIndex vrNextVariant  ; /* ptr to next variant */
      typeIndex vrFields  ;     /* fields for this variant */
    )  vrRecdInfo ;

struct {    /* set type */
      boolean   stPacked ;   /* packed set */
      unsigned  stCount  ;  /* number of elements */
      typeIndex stBase  ;   /* ptr to base type of set */
      constDesc stFirst, stLast  ; /* set bounds */
    ) setInfo  ;
```


Pascal Example - Type Table Entry (continued)

```
    struct { /* file type */
        boolean   fPacked  ;  /*  packed file */
        typeIndex fComponent ;   /* file object type */
        ) ;
    typeIndex  pObject  ;   /*  pointer type */
    symIndex   fwdName  ;   /*  forward type */
  } tyInfo ;       /*  end tyKind union */
}  typeEntry ;
```

Pascal Example - Symbol Table Entry, Tables

```
/* definition of symbol table entry */
typedef struct symStruct {
      name    syName  ;    /* symbol's name */
      symKind syKind ;     /* kind of symbol */
      typeIndex syType  ;  /* ptr to type table */
      symIndex  syNext ;   /* link to related symbol */
      lAddress  syLAddr ;  /* logical address */
      /* fields dependent on kind of symbol */
      union {
          constDesc : syCValue ;    /* constant's value */
          struct {     /* variable */
              mAddress vAddress  ;  /* run time address */
              unsigned vSize  ;     /* size */
            }  syVarInfo ;
          pfDesc pfInfo ;      /*  procedure/function */
          mAddress lblAddr  ;     /*  statement label */
      } syInfo ;
}  symEntry ;

/*  Table Definitions */
char * stringTable[ maxString ] ;   /*  String Table */
symEntry   syt[ maxSymbols ] ;      /*  Symbol Table */
typeEntry  tyt[ maxTypes ;          /*  Type Table   */
```


food for slide eater

food for slide eater

food for slide eater