Naming, Scoping, and Binding

LR Parsers

- SLR(1) – LR(0) items (construct DFA for recognizing viable prefix with no lookahead), use FOLLOW information to guide reductions
- LALR(1) – merge states of LR(1) with same core (LR(0) items)
- LR(1) – store lookahead information in DFA
- LR(k)

More general

Types of Conflicts

- Ambiguous grammar constructs resulting in multiple actions in action table
  - Shift/reduce (S←Ay, A←x|xy)– can opt in favor of longest match, i.e., shift, or modify grammar to eliminate it – classic example is dangling else
  - Reduce/reduce – (S←Ax|Bxy, A←x, B←x) - often no simple solution

Parsing Review

	Advantages	Disadvantages
Recursive descent or predictive LL(1)	Fast Simple Automatable	No left recursion LL(1) ⊂ LR(1)
LR(1)	Fast Automatable	Larger table size Right recursion can be inefficient

- Name – a character string or identifier used to represent something
  - Provide a level of abstraction – control (e.g., subroutine) or data (classes)
- Binding – an association between a name and the thing it names
- Scope – textual region in the program in which a binding is active
  - The set of active bindings is called the current referencing environment

Binding Time

- The time at which a binding is created (time at which implementation decision is made)
  - Language design time
  - Language implementation time
  - Program writing time
  - Compile time
  - Link time
  - Load time
  - Run time

Static binding – before run time
Dynamic binding – at run time

Object Lifetime and Storage Management

- Events
  - Creation of objects
  - Creation of bindings
  - References to variables, subroutines, types, etc., all of which use bindings
  - Deactivation and reactivation of bindings
  - Destruction of bindings
  - Destruction of objects

Binding's lifetime –

Period of time between creating and destruction of name-to-object binding

Storage Allocation Mechanisms

- Static objects – given an absolute address since they are retained throughout the program
- Stack objects – allocated in LIFO order, usually in conjunction with subroutine calls and returns
- Heap objects – allocated and deallocated at arbitrary times – more general and expensive storage management

Scope Rules

- Scope
  - a program region of maximal size in which no bindings change
  - name space that maps a set of names to a set of variables
- Scope of a binding
  - Static – can be determined based purely on textual rules at compile time
  - Dynamic – depends on the flow of execution at run time
- Lifetime of a variable – single execution of the scope or multiple executions of the scope (decides where data can be allocated)

Static Scope

- Only a single global scope, variables declared by virtue of being used - Basic
- Global and local scope – e.g., C, Fortran (variable declaration optional (assumed to be local), "common blocks" used to "import" global variables into subroutines)
- Closest lexically nested scope rule (nested subroutines) – e.g., Pascal (Algol family)
- Modules (collection of objects - subroutines, variables, types, etc.) with internals/externals invisible unless explicitly exported/imported – inactive bindings outside (not destroyed) – e.g., separate compilation facilities of C, namespaces in C++
  - Closed scope – requiring explicit import (Modula)
  - Open scope – one not requiring explicit import (Ada, nested subroutines)
    - All scopes surrounding the current scope
- Module types and classes – multiple instances, requiring creation/initialization and destruction, specification of instance – Simula, Euclid
  - Class – object-oriented programming construct, includes inheritance, e.g., Smalltalk, Eiffel, C++, Java

Dynamic Scope

- Depends on the order in which subroutines are called – current binding is the most recent encountered during execution – e.g., Perl, APL, Snobol

Shallow vs. Deep Binding

- References to subroutines (e.g., passed as parameter) can have scope rules applied
  - When reference is first created (deep)
  - When routine is finally called (shallow)

Overloading

- Name that can refer to more than one object in a given scope is said to be overloaded, e.g., addition operator (+) used for integer and floating point addition
- Lookup routine of symbol table must return list of possible meanings for the requested name
- Semantic analyzer makes choice based on context (number and types of arguments, for example)
- Distinguish between overloading, coercion, polymorphism, generics
  - Coercion – automatic conversion by compiler from one type to another
  - Polymorphic parameters – represent unconverted arguments of several types with common characteristics (e.g., counting elements in a linked list) – single object capable of accepting multiple types
  - Generic subroutines – template that can be used to create multiple concrete subroutines that differ in minor ways – multiple objects that accept arguments of different types

Aliasing

- More than one name for the same thing – save space
  - Fortran – use common blocks in different ways (equivalence statement) – improve code efficiency
  - C unions – multiple representations

Modules

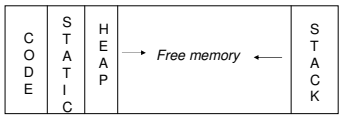
- Collection of subroutines, variables, types, etc. that are visible to each other but invisible outside unless explicitly exported
  - Clu, Modula, Turing, Ada
  - Separate compilation in C
    - Import data/functions using "extern"
    - "static" outside a function means it is usable only inside the current source file
- Closed (Modula) versus open (Ada) scopes
- Modules as types to allow multiple instances – Simula, Euclid
- Classes – module types augmented with inheritance – Smalltalk, Eiffel, C++, Java

Procedures

- Control abstraction
  - Well-defined entry/exits
  - Mechanism to pass parameters, return values
- Name space/scope
  - New name space within procedure
  - Local names are protected from outside
- External interface
  - Accessed by procedure name, parameters
  - Protection for both caller and callee
  - Enables software libraries, systems
- Separate compilation
  - Compile procedures independently
  - Keeps compile times reasonable
  - Allows us to build large programs

Run-Time Storage Organization

Typical memory layout



→Allows both stack and heap maximal freedom

Scoping Rules of Languages

- LISP
  - Dynamic scoping
  - Most recently invoked procedure
- Algol, Pascal, Modula
  - Nested lexical scoping
  - Procedures nested within procedures
- Fortran 77
  - Global scope – common blocks
  - Local scope – variables, parameters in procedures

Scoping Rules of Languages

- C
  - Global scope – procedures, external variables
  - File scope – variables declared in files
  - Procedure scope – local variables
  - Nested block scope – within {}
- Java
  - Global scope – public classes
  - Package scope – fields, methods within package
  - Procedure scope – local variables
  - Nested block scope – within {}

Symbol Table (for static scoping)

- Maps names to the information the compiler knows about them
- Basic operations – insert, lookup
- Each scope assigned a serial number
  - Predefined identifiers assigned 0
  - Global scope assigned 1
  - Additional scopes given successive numbers as they are encountered
- All names regardless of scope entered in a single large hash table, keyed by name
  - Entry contains symbol name, category, scope number, type, etc.
- Scope stack to maintain current referencing environment
  - Used by semantic analyzer

Association Lists and Central Reference Tables (Dynamic Scope)

- Association list – list of name/value pairs (e.g., Lisp)
  - Functions as a stack
  - Search from front of list until appropriate binding is found (inefficient)
- Central reference table – list (stack) of entries for each distinct name in the program, most recent occurrence at top
  - Lookup operations faster

Scanning Complexities of Real Programming Languages

- Fortran 66 and 77 – blanks not significant
- PL/I – no reserved keywords
- C++ - template definition
  - PriorityQueue<MyType<int>>
    - >> is a C++ operator – distinction requires coordination between scanner and parser

→ Sound theoretical basis for scanning probably influenced language design in a positive way!

Parsing Complexities in Real Languages

- Using one word to represent two different meanings – ambiguous grammar
  - E.g., function and array reference in Fortran
- Left versus right recursion
  - Top-down parsers need right recursive grammars
  - Bottom-up parsers can accommodate both, however, right recursive grammar requires more stack space
- Associativity –
  - Left recursion naturally produces left associativity
  - Right recursion naturally produces right associativity

Naming Complexities in Language Design

- Recursive subroutine cannot declare a local object with the same name as the subroutine itself – problem in Pascal that returns values by assigning to function name
- External declarations in C