A **lexical analyzer** is a pattern matcher that isolates the small-scale parts of a program, which are called lexemes. **Lexemes** occur in categories, such as integer literals and names. These categories are called **tokens**. Each token is assigned a numeric code, which along with the lexeme is what the lexical analyzer produces. There are three distinct approaches to constructing a **lexical analyzer**: using a software tool to generate a table for a table-driven analyzer, building such a table by hand, and writing code to implement a state diagram description of the tokens of the language being implemented. The **state diagram** for tokens can be reasonably small if character classes are used for transitions, rather than having transitions for every possible character from every state node. Also, the state diagram can be simplified by using a table lookup to recognize **reserved words**.

Chap5

A **reserved word** is a special word of a programming language that cannot be used as a name. There is one potential problem with reserved words: If the language includes a large number of reserved words, the user may have difficulty making up names that are not reserved. The **address** of a variable is the machine memory address with which it is associated.

A binding is **static** if it first occurs before run time begins and remains unchanged throughout program execution. If the binding first occurs during run time or can change in the course of program execution, it is called **dynamic.**

An **explicit declaration** is a statement in a program that lists variable names and specifies that they are a particular type. **An implicit declaration** is a means of associating variables with types through default conventions, rather than declaration statements. Implicit variable type binding is done by the language processor, either a compiler or an interpreter. There are several different bases for implicit variable type bindings. The simplest of these is naming conventions. Some of the problems with implicit declarations can be avoided by requiring names for specific types to begin with particular special characters.

The memory cell to which a variable is bound somehow must be taken from a pool of available memory. This process is called **allocation**. **Deallocation** is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory.

**Static variables** are those that are bound to memory cells before program execution begins and remain bound to those same memory cells until program execution terminates. One advantage of static variables is efficiency. One disadvantage of static binding to storage is reduced flexibility.

**Stack-dynamic variables** are those whose storage bindings are created when their declaration statements are elaborated, but whose types are statically bound. **Elaboration** of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached. Therefore, elaboration occurs during run time.

The **advantages of stack-dynamic variables** are as follows: To be useful, at least in most cases, recursive subprograms require some form of dynamic local storage so that each active copy of the recursive subprogram has its own version of the local variables. These needs are conveniently met by stack-dynamic variables. The **disadvantages**, relative to static variables, of stack-dynamic variables are the run- time overhead of allocation and deallocation, possibly slower accesses because indirect addressing is required, and the fact that subprograms cannot be history sensitive.

**Explicit heap- dynamic variables** are nameless (abstract) memory cells that are allocated and deallocated by explicit run-time instructions written by the programmer. **The disadvantages** of explicit heap-dynamic variables are the difficulty of using pointer and reference variables correctly, the cost of references to the variables, and the complexity of the required storage management implementation.

**Implicit heap-dynamic variables** are bound to heap storage only when they are assigned values.

The **advantage** of such variables is that they have the highest degree of flexibility, allowing highly generic code to be written. One **disadvantage** of implicit heap- dynamic variables is the run- time overhead of maintaining all the dynamic attributes, which could include array subscript types and ranges, among others. Another disadvantage is the loss of some error detection by the compiler.

The **referencing environment** of a statement is the collection of all variables that are visible in the statement.

Chap6

A **rectangular array** is a multidimensional array in which all of the rows have the same number of elements and all of the columns have the same number of elements. Rectangular arrays model rectangular tables exactly. A **jagged array** is one in which the lengths of the rows need not be the same. A **slice** of an array is some substructure of that array.

A **static array** is one in which the subscript ranges are statically bound and storage allocation is static (done before run time). The advantage of static arrays is efficiency: No dynamic allocation or deallocation is required. The disadvantage is that the storage for the array is fixed for the entire execution time of the program.

**A fixed heap-dynamic array** is similar to a fixed stack-dynamic array, in that the subscript ranges and the storage binding are both fixed after storage is allocated. The differences are that both the subscript ranges and storage bindings are done when the user program requests them during execution, and the storage is allocated from the heap, rather than the stack. The advantage of fixed heap-dynamic arrays is flexibility—the array’s size always fits the problem. The disadvantage is allocation time from the heap, which is longer than allocation time from the stack.

A **heap-dynamic array** is one in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array’s lifetime. The advantage of heap-dynamic arrays over the others is flexibility: Arrays can grow and shrink during program execution as the need for space changes. The disadvantage is that allocation and deallocation take longer and may happen many times during execution of the program.

Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point. For example, the number 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type. The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful, for reasons discussed in the following paragraph.

Chap4 HW

*What are three reason why syntax analyzer are based on grammars?* The grammar description can be use as a direct basis for the analyzer, implementations based on BNF are relatively easy to maintain because of their modularity and Grammars are clear and concise.

*Describe briefly the three approaches to building a lexical analyzer.*

Write a formal description of the token patterns of the language using a descriptive language related to regular expression. Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram. Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

*Perform the pairwise disjointness test for the following grammar rules.*

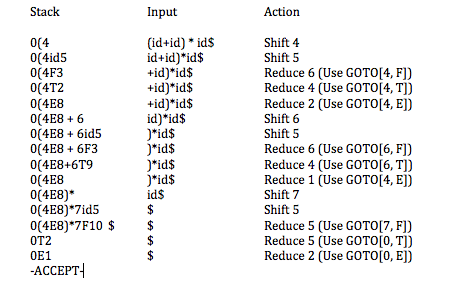
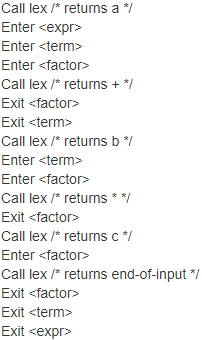
|  |  |  |
| --- | --- | --- |
| S→ aSb | bAA  first(aSb) = a  first(bAA) = bare not intersected, the rule passes  A → b{aB} | a  first(b{aB}) = b  first(a) = a  are not intersected, the rule passes | B -> aB | ba | aBb  first(aB) = a  first(ba) = b  first(aBb) = a  They are intersected & thus the rule fails the test. | C -> aaA | b | caB  first(aaA) = a  first(b) = b  first(caB) = c  are not intersected & thus the rule passes |

*Construction of parse tree given grammar S → AbB | bAc ----- A→Ab | aBB ---- B → Ac | cBb | c*

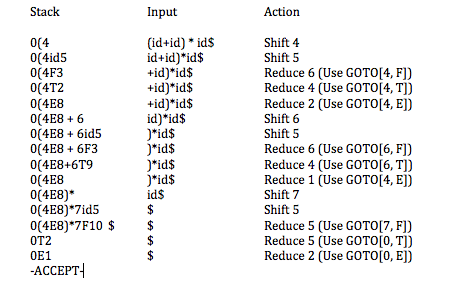
|  |  |  |
| --- | --- | --- |
|  |  |  |

*Write an EBNF rule that describes the for statement of Java or C++. Write the recursive-descent subprogram in Java or C++ for this rule. Assume the following non-terminals are given: <type>, <id>, <literal>, <assign>, <expr>, and <stmt\_list>[xuong hang]*<for> -> for ‘(‘ [[<type>] <id> = <expr> {, [<type>] <id> = <expr>}] ; [<expr>] ; [<expr> {, <expr>}] ‘)’ ‘{‘ <stmt\_list> ‘}’

*Show a trace of the recursive descent parser given in Section 4.4.1 for the string a + b \* c*

(id + id) \* id

*Show a complete parse, including the parse stack contents, input string, and action for the string id \* (id + id), using the grammar and parse table in Section 4.5.3.*



*There are 2 design issues for names:*

-are name case sensitive? -are the special words of language reserved words or keywords?

*In what way are reserved words better than keywords?*

A reserved word is a special word that cannot be used as a user-defined name. A keyword is a word that is special only in certain contexts.

*Some programming languages are typeless. What are the obvious advantages and disadvantage of having no types in language?*

Advantages: allow users to write sloppy programs faster.

Disadvantages: cannot control the data and variables, compiler cannot detect any mistakes. that its not good programming habit and not accepted by powerful languages (c/c++/java). It allows programmer to make mistake easily.

*Write a simple assignment statement with one arithmetic operator in some language you know. For each component of the statement, list the various bindings that are required to determine the semantics when the statement is executed. For each binding, indicate the binding time used for the language*

int count; count = count + 5;

*Describe a situation when a history-sensitive variable in a subprogram is useful.*

Answer: Globally accessible variables are often used throughout the execution of a program, thus making it necessary to have them bound to the same storage during that execution. Sometimes it is convenient to have subprograms that are history sensitive.

Chap6 HW

*What are the design issues for character sting types?*

Should strings be simply a special kind of character array or a primitive type? Should strings have static or dynamic length?

*Describe the three string length options.*  
**Static length string**: the length that can be static and set when the string is created. This is the choice for strings of Pyhthon, the immutable objects of Java’s string class, as well as similar classes in the C++ standard class library, Ruby’s built-in string class, and the .NET class library available to C# and F#.  
**Dynamic length**: allow strings to have varying length with no maximum. need run-time descriptor; allocation/de-allocation is the biggest implementation problem  
**Limited dynamic length**: allow strings to have varying length up to a declared and fixed maximum set by the variable’s definition. may need a run-time descriptor for length (but not in C and C++)

*Define ordinal, enumeration, and subrange types.*

- Ordinal type is one in which the age of possible values can be easily associated with the set of positive integers. In Java, for example, the primitive ordinal types are integer, char, and Boolean.

- Enumeration types is an all possible values, which are named constants, are provided in the definition.

- Subrange type is a contiguous subsequence of an ordinal type.

*What are the arguments for and against representing Boolean values as single bits in memory?*

For: Using a single bit instead of an entire byte will conserve memory so boolean variables stored as single bits are very space efficient

*Compare the tombstone and lock-and-key methods of avoiding dangling pointers, from the point of view of safety and implementation cost.*

Answer: Tombstones take more memory (for the tombstones).

kock-&-key requires add’l cpu time on each ptr assignment to copy key as well as pointer. Pointer arithmetic could overwrite key in the heap.

*Define row major order and column major order.*

3 4 7

6 2 5

1 3 8

Row major order (by rows): the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth.

It would stored in row major order as: 3, 4, 7, 6, 2, 5, 1, 3, 8

Column major order (by columns): the elements of an array that have as their last subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the last subscript, and so forth.

It would stored in row major order as: 3, 6, 1, 4, 2, 3, 7, 5, 8

*Chapter7*

The **operator precedence rules** for expression evaluation partially define the order in which the operators of different precedence levels are evaluated. The operator precedence rules for expressions are based on the hierarchy of operator priorities, as seen by the language designer. The operator precedence rules of the common imperative languages are nearly all the same, because they are based on those of mathematics.

When an expression contains two adjacent occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the **associativity** rules of the language. An operator can have either left or right **associativity**, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.

Languages that allow **parentheses** in arithmetic expressions could dispense with all precedence rules and simply associate all operators left to right or right to left.

A **side effect** of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable. (A global variable is declared outside the function but is accessible in the function.)

A number of errors can occur during expression evaluation. If the language requires type checking, either static or dynamic, then operand type errors cannot occur. The errors that can occur because of coercions of operands in expressions have already been discussed. The other kinds of errors are due to the limitations of computer arithmetic and the inherent limitations of arithmetic. The most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored. This is called **overflow** or **underflow**, depending on whether the result was too large or too small. One **limitation of arithmetic** is that division by zero is disallowed.

A **relational operator** is an operator that compares the values of its two operands. A relational expression has two operands and one relational operator. The relational operators are often overloaded for a variety of types. The operation that determines the truth or falsehood of a relational expression depends on the operand types.

A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators.

Chap8

**Selection statements** fall into two general categories: two-way and n-way, or multiple selection.

A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained. It also includes some means of specifying the initial and terminal values of the loop variable, and the difference between sequential loop variable values, often called the **stepsize**. The **initial**, **terminal**, and stepsize specifications of a loop are called the loop parameters

The **iterator** is called at the beginning of each iteration, and each time it is called, the iterator returns an element from a particular data structure in some specific order.

Chapter9

There are two ways that a nonmethod subprogram can gain access to the data that it is to process: through **direct access to nonlocal variables** (declared elsewhere but visible in the subprogram) or **through parameter passing**.

The parameters in the subprogram header are called **formal parameters**. They are sometimes thought of as dummy variables because they are not variables in the usual sense: In most cases, they are bound to storage only when the subprogram is called, and that binding is often through some other program variables.

Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called **actual parameters**. They must be distinguished from formal parameters, because the two usually have different restrictions on their forms, and of course, their uses are quite different.

The **disadvantage** to keyword parameters is that the user of the subprogram must know the names of formal parameters.

There are two distinct categories of subprograms—procedures and functions— both of which can be viewed as approaches to extending the language. **Subprograms** are collections of statements that define parameterized computations. **Functions** return values and procedures do not.

An **overloaded** subprogram is one that has the same name as another subprogram in the same referencing environment. A **generic** subprogram is one whose computation can be done on data of different types in different calls.

The **primary advantage of static local variables over stack-dynamic loca**l variables is that they are slightly more efficient—they require no run-time over - head for allocation and deallocation.

When a parameter is **passed by value**, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram, thus implementing in-mode semantics. Pass-by- value is normally implemented by copy, because accesses often are more efficient with this approach. It could be implemented by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell.

The advantage of **pass-by-value** is that for scalars it is fast, in both linkage cost and access time. The main disadvantage of the pass-by-value method if copies are used is that additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram.

The pass-by-result method has the advantages and disadvantages of passby- value, plus some additional disadvantages. If values are returned by copy (as opposed to access paths), as they typically are, pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value. As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by copy.

**Pass-by-value-result** is an implementation model for inout-mode parameters in which actual values are copied. It is in effect a combination of pass-by-value and pass-by-result.

**Pass-by-reference** is a second implementation model for inout-mode parameters. The advantage of pass- by- reference is that the passing process itself is efficient, in terms of both time and space. Duplicate space is not required and no copying is required.

An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment.

If a static- scoped programming language does not allow nested subprograms, **closures** are not useful, so such languages do not support them. All of the variables in the referencing environment of a subprogram in such a language (its local variables and the global variables) are accessible, regardless of the place in the program where the subprogram is called.

A **coroutine** is a special kind of subprogram. Rather than the master- slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable. In fact, the coroutine control mechanism is often called the **symmetric unit control** model. **Coroutines** can have multiple entry points, which are controlled by the coroutines themselves. They also have the means to maintain their status between activations. This means that coroutines must be history sensitive and thus have static local variables. Secondary executions of a coroutine often begin at points other than its beginning. Because of this, the invocation of a coroutine is called a resume rather than a call.

Typically, coroutines are created in an application by a program unit called the master unit, which is not a coroutine. When the entire family of coroutines is constructed, the master program resumes one of the coroutines, and the members of the family of coroutines then resume each other in some order until their work is completed, if in fact it can be completed. If the execution of a coroutine reaches the end of its code section, control is transferred to the master unit that created it. This is the mechanism for ending execution of the collection of coroutines, when that is desirable. In some programs, the coroutines run whenever the computer is running.

Chapter10

A **subprogram** call in a typical language has numerous actions associated with it. The call process must include the implementation of whatever parameter- passing method is used.

The semantics of a call to a “simple” **subprogram** requires the following actions: 1. Save the execution status of the current program unit. 2. Compute and pass the parameters. 3. Pass the return address to the called. 4. Transfer control to the called.

The semantics of a return from a simple subprogram requires the following actions: 1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters. 2. If the subprogram is a function, the functional value is moved to a place accessible to the caller. 3. The execution status of the caller is restored. 4. Control is transferred back to the caller.

The call and return actions require storage for the following: • Status information about the caller • Parameters • Return address • Return value for functions • Temporaries used by the code of the subprograms.

A simple **subprogram** consists of two separate parts: the actual code of the subprogram, which is constant, and the local variables and data listed previously, which can change when the subprogram is executed.

Activating a subprogram requires the dynamic creation of an instance of the activation record for the subprogram. As stated earlier, the format of the activation record is fixed at compile time, although its size may depend on the call in some languages. Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to create instances of these activation records on a stack. This stack is part of the runtime system and therefore is called the **run-time stack**, although we will usually just refer to it as the stack.

One of the most important advantages of stack- dynamic local variables is support for recursion. Therefore, languages that use stack- dynamic local variables also support recursion.

The format, or layout, of the noncode part of a subprogram is called an activation record, because the data it describes are relevant only during the activation or execution of the subprogram. The form of an activation record is static. An activation record instance is a concrete example of an activation record, a collection of data in the form of an activation record.

The **caller actions** are as follows: 1. Create an activation record instance. 2. Save the execution status of the current program unit. 3. Compute and pass the parameters. 4. Pass the return address to the called. 5. Transfer control to the called.

The **epilogue actions** of the called are as follows: 1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters. 2. If the subprogram is a function, the functional value is moved to a place accessible to the caller. 3. Restore the stack pointer by setting it to the value of the current EP minus one and set the EP to the old dynamic link. 4. Restore the execution status of the caller. 5. Transfer control back to the caller.

The collection of dynamic links present in the stack at a given time is called the **dynamic chain**, or **call chain**. It represents the dynamic history of how execution got to its current position.

Chapter11

The concept of **process abstraction** is among the oldest in programming language design (Plankalkül supported process abstraction in the 1940s). All subprograms are process abstractions because they provide a way for a program to specify a process, without providing the details of how it performs its task (at least in the calling program).

**Floating- point** types in high- level languages employ a key concept in data abstraction: information hiding.

The **getters** and **setters** of instance variables can be used in two ways, either in method calls or in dot notation, as if they were publically accessible.

In languages that allow nested subprograms, programs can be organized by nesting subprogram definitions inside the logically larger subprograms that use them. This can be done in Python and Ruby.

Java data abstractions are similar to those of C++, except all Java objects are allocated from the heap and are accessed through reference variables.

To support the construction of large programs, some contemporary languages include multiple- type encapsulation constructs, which can contain a collection of logically related types. An encapsulation may also provide access control to its entities. Encapsulations provide the programmer with a method of organizing programs that also facilitates recompilation.

Chapter13

**Concurrency** in software execution can occur at four different levels: instruction level (executing two or more machine instructions simultaneously), statement level (executing two or more high-level language statements simultaneously), unit level (executing two or more subprogram units simultaneously), and program level (executing two or more programs simultaneously).

A run-time system program called a **scheduler** manages the sharing of processors among the tasks. If there were never any interruptions and tasks all had the same priority, the scheduler could simply give each task a time slice, such as 0.1 second, and when a task’s turn came, the scheduler could let it execute on a processor for that amount of time.

In the environment of sequential programs, a program has the **liveness** characteristic if it continues to execute, eventually leading to completion.

suppose task A and task B both need the shared resources X and Y to complete their work. Furthermore, suppose that task A gains possession of X and task B gains possession of Y. After some execution, task A needs resource Y to continue, so it requests Y but must wait until B releases it. Likewise, task B requests X but must wait until A releases it. Neither relinquishes the resource it possesses, and as a result, both lose their liveness, guaranteeing that execution of the program will never complete normally. This particular kind of loss of liveness is called **deadlock**.

A **semaphore** is a simple mechanism that can be used to provide synchronization of tasks. a **semaphore** is a data structure that consists of an integer and a queue that stores task descriptors. A **task** **descriptor** is a data structure that stores all of the relevant information about the execution state of a task.

A semaphore that requires only a binary-valued counter, like the one used to provide competition synchronization, is called a b**inary semaphore**.

Accesses to objects of the class are controlled by adding the **synchronized** modifier to the access methods.

A thread can be terminated with the **Abort** method, although it does not literally kill the thread.

the **AtomicInteger** class defines getter and setter methods, as well as methods for add, increment, and decrement operations. These operations are all atomic.

1. **Explain clearly why a race condition can create problems for a system.**

Because two or more tasks are racing to use the shared resource and the behavior of the program depends on which task arrives first (and wins the race). The importance of competition synchronization should now be clear.

**What are the different ways to handle deadlock?**

When deadlock occurs, assuming that only two program units are causing the deadlock, one of the involved program units should be gracefully terminated, thereby allowed the other to continue.

24. **What is an overloaded subprogram?**Answer : Overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing environment. Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its parameters, and possibly in its return type if it is a function.

26. **What is multicast delegate?**

Answer : Multicast delegate is all of the methods stored in a delegate instance are called in the order in which they were placed in the instance.

