A state transition diagram, or just **state diagram**, is a directed graph. State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called **finite automata**. Finite automata can be designed to recognize members of a class of languages called regular languages. Regular grammars are generative devices for **regular languages**. The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

A **recursive-descent parser** is a coded version of a syntax analyzer based directly on the BNF description of the syntax of language.

A **bottom- up parser** constructs a parse tree by beginning at the leaves and progressing toward the root. This parse order corresponds to the reverse of a rightmost derivation. That is, the sentential forms of the derivation are produced in order of last to first. The process of finding the correct RHS to reduce is complicated by the fact that a given right sentential form may include more than one RHS from the grammar of the language being parsed. The correct RHS is called the **handle**. A right sentential form is a sentential form that appears in a rightmost derivation.

Every parser for a programming language is a **pushdown automaton** (**PDA**), because a PDA is a recognizer for a context-free language. A PDA is a very simple mathematical machine that scans strings of symbols from left to right.

The only disadvantage of **LR parsing** is that it is difficult to produce by hand the parsing table for a given grammar for a complete programming language.

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**.

**Syntax analyzers** have two goals: to detect syntax errors in a given program and to produce a parse tree, or possibly only the information required to build such a tree, for a given program. Parsers that work for all unambiguous grammars have complexity O(n3 ). However, parsers used for implementing syntax analyzers for programming languages work on subclasses of unambiguous grammars and have complexity O(n).

Two distinct **grammar characteristics** prevent the construction of a recursive-descent parser based on the grammar. One of these is left recursion. The process of eliminating direct left recursion from a grammar is relatively simple. The other problem is detected with the pairwise disjointness test, which tests whether a parsing subprogram can determine which RHS is being parsed on the basis of the next token of input.

Chap5 **Names** in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters ( \_ ).

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.

*Explain the three reason why lexical analysis is separated from syntax analysis.* Simplicity-Techniques for lexical analysis are less complex than those required for syntax analysis, so the lexical-analysis process can be simpler if it is separate. Efficiency-Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time, it is not fruitful to optimize the syntax analyzer. Portability-Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent. However, the syntax analyzer can be platform independent. It is always good to isolate machine-dependent parts of any software system.

*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.*

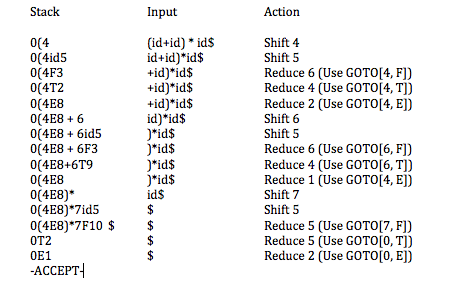
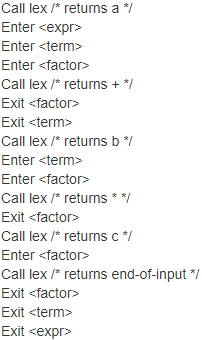
|  |  |  |
| --- | --- | --- |
| S→ aSb | bAA  first(aSb) = a  first(bAA) = bare not intersected, the rule passes  A → b{aB} | a  first(b{aB}) = b  first(a) = a  are not intersected, the rule passes | B -> aB | ba | aBb  first(aB) = a  first(ba) = b  first(aBb) = a  They are intersected & thus the rule 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*

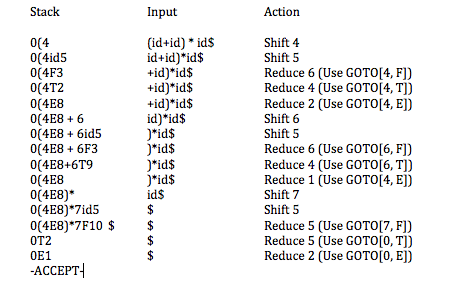
|  |  |  |
| --- | --- | --- |
|  |  |  |

*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.

Chap 3 HW

*Write EBNF descriptions for the following:*  
 C union definition

= <union\_defn> -> union <var\_list> <union\_identifier>;

<var\_list> -> <list\_of\_data-type specifier> <var>

<list\_of\_data-type specifier> -> int | float | long |char | double

<union\_identifier> -> <var>

*Rewrite the BNF to give + precedence over \* and force + to be right associative.*

Answer:

<assign>-> <id> = expr

<id> -> A| B| C

<expr>-> <expr> -<term>

|<term>

<term>-> <term> / <factor>

| <factor>

<factor> -> (<expr>)

|<id>

*Write a BNF description of the Boolean expressions of Java, including the three operators &&, ||, and ! and the relational expressions.*

<Boolean\_expr> → <Boolean\_expression> ||<Boolean\_term> | <Boolean\_term>

<Boolean\_term> → <Boolean\_term> && <Boolean\_factor> | <Boolean\_factor>

<Boolean\_factor> → id | ! <Boolean\_factor> | (<Boolean\_expr>) | <relation\_expr>

<relation\_expr> → id == id | id !=id | id < id | id <= id| id >= id | id > id

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