CS 447 Compiler Design, Fall 2011

Compiler Organization

A compiler is a complex piece of software. At the highest level, it translates a program written in one language (called the source program) to a second language (called the target program). This target program may be an executable machine language for the machine the compiler is running on, in which case it is called a native code compiler, or it may be an executable program for a different machine, in which case it is called a cross compiler. Or, it may be an assembly language or it may be some other type of intermediate language.

The compiling process goes through several stages. See the overhead slide of figure 1.1.

- Scanner or Lexical Analyzer
- Parser or Syntactical Analyzer
- Checker or Semantic Analyzer
- Source Code Optimizer
- Code Generator
- Target Code Optimizer or Peep-Hole Optimizer

There are also several major data structures involved

- Tokens
- Abstract Syntax Tree
- Annotated or Decorated Syntax Tree (Attribute Grammar)
- Literal Table
- Symbol Table

Other Issues in Compiler Structure

Front-end and Back-end
Passes
Pragmas or Compiler directives

Introduction: Formal Languages

A formal language is one where we are only concerned about the syntax of an expression and not the semantics. For example, the two sentences (a) *Loudly barked the startled dog* and (b) *Furiously slept the green fog* both have the same syntax and are both equally valid as expressions in a formal language. However, the semantics or meaning is quite different.

In order to recognize tokens, a scanner must perform a pattern-matching operation. The elements that form tokens are usually part of a *regular* language. That is, a language whose elements are formed by *regular expressions*.

What is a regular expression?

A regular expression is one of the following

A basic regular expression consists of a single character **`a`**, where **`a`** is from an alphabet Σ of legal characters, the metacharacter ε or the metacharacter ϕ.

An expression of the form **`r|s`** where both **`r`**and **s** are regular expressions and | represents choice.

An expression of the form **`rs`**, where both **r** and **s** are regular expression and the operation is concatenation.

An expression of the form **`r*`**, where **r** is a regular expression and ***** is the Kleene (Stephen) closure, meaning 0 or more repetitions of **`r`**.

An expression of the form **`(r),`** where **r** is a regular expression. The parentheses do not change the language; they only adjust the precedence of the operations.

What is the language defined by a regular expression?

Example: **`a ( a | b )* b`**
This is the language of strings that begin with an **a** and end with a **`b`**. In between, there can be any number of **`a`**'s or **`b`**'s in any order.

Meta characters: **`(`**, **`)`** , **`|`**, **`*`**

Extensions to regular expressions

One or more repetitions: +

Any character: •

Character classes or ranges of characters
0 | 1 | … | 9 or **[**0-9**]**
a | b | c | … | z | A | B | C | … | Z or **[**a-zA-Z**]**

Any character not in a class ~(a | b | c) or **[**^abc**]**

0 or 1 characters: **?**

Naming subclasses: *Natural* = [0-9]+ *signedNatural* = (+ | –)? *Natura*l

Recognizing regular expression: Finite Automata

A Finite Automaton (FA) is one of the machines in the Chomsky Hierarchy. It has a formal mathematical definition.

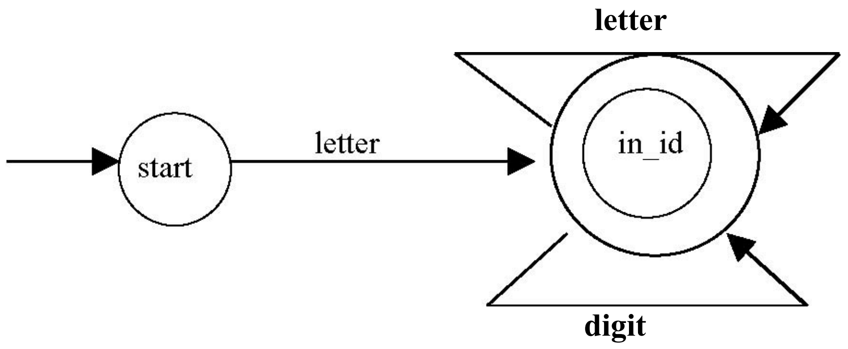
An FA = { Q, F, Σ, δ }, where
- $Q = \{ q_0, q_1, \ldots, q_n \}$ is a set of states and q_0 is the start state.
- $F \subseteq Q$ is the set of final states
- $\Sigma = \{ x_1, x_2, \ldots, x_m \}$ is the alphabet
- $\delta(q_i, x_j) = q_k$ is a transition function

An FA is also depicted graphically where a state is represented by a circle and a transition is represented by an arrow from one state to another. The arrow is labeled by the character from the alphabet that caused the transition. That is, given the transition

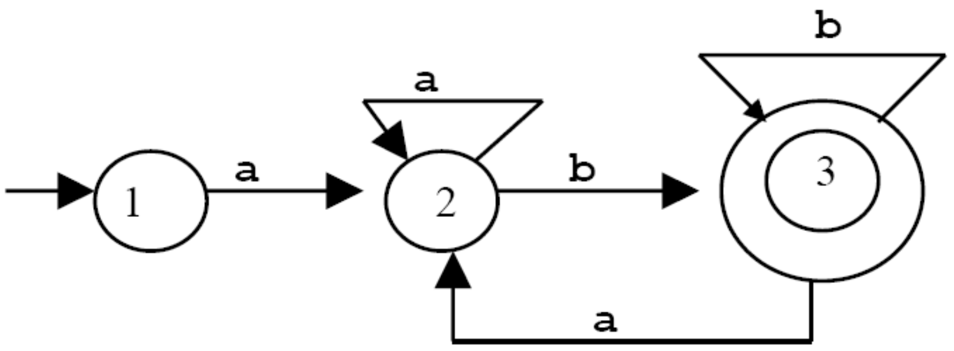
$\delta(q_i, x_j) = q_k$, there would be a circle representing q_i, a circle representing q_k, and an arrow going from q_i to q_k labeled with x_j.

As an example, consider a regular expression for an identifier.

```
letter ( letter | digit )*
```



Consider the expression: **`a (a | b )* b`**



This is deterministic because for every state and every letter of the input alphabet, there is one and only one state to go to.

One way to implement a DFA is by a table

	a	**b**	**$**	**Accept**
1	2			no
2	2	3		no
3	2	3	3	yes

Implementing a DFA by a table is how automated scanner generators such as Lex work.

	a	**b**	**$**
1	2	4	4
2	2	3	4
3	2	3	3
4	4	4	4

T

1	F
2	F
3	T
4	F

Accept

1	F
2	F
3	F
4	T

Error

1	T
2	T
3	T
4	F

Advance

Use state 4 as an **error** state. In the table above, the empty spaces would be replaced by a 4.

Assume that the start state is state 1

```
   state = 1
   input = getInputChar();


   while( ! (Accept[ state ] && Error[ state ] )) {
       newState = T[ state, input ]
        if( Advance[ newState ] )
           input = getInputChar();
        state = newState;
      }
      if( Accept[ state ] ) then
         accept();
      else
          error();
```

In this scheme, we assume the table **T** is indexed by a state and a character and that there are 3 Boolean arrays indexed by states: `Accept[]`, `Error[]`, and `Advance[]`.

A second method, which is normally used when coding by hand, is the following

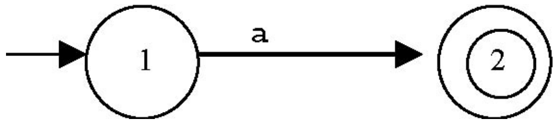
```
done = false;
state = 1;
input = getInputChar();

while( ! done ) {
     {switch( state ) {
        case 1:
           switch( input ) {
             case a: state = 2; input = getInputChar(); break;
             default: state = error; done = true; break;
        }

        case 2:
           switch( input ) {
              case a: input = getInputChar(); break;
              case b: state = 3; getInputChar(); break;
              default: state = error; done = true; break;
        }

        case 3:
           switch( input ) {
               case a: state = 2; input = getInputChar(); break;
               case b: input = getInputChar(); break;
               case $: done = true; break;
               default: state = error; done = true; break;
        }
}

if( state == 3 )
    accept();
else
    error();
```

Thus, if we have a DFA, it is relatively straightforward to write a scanner. The question is, "How do we create a DFA?"

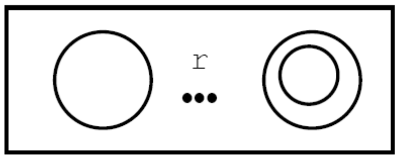
Given a regular expression, we can create a DFA using a two-step process. In the first step, we use Thompson's Construction to create a NFA, and in the second step, we use the Subset Construction to convert the NFA to a DFA.

Thompson's Construction

For each letter **a**εΣ, including ε,

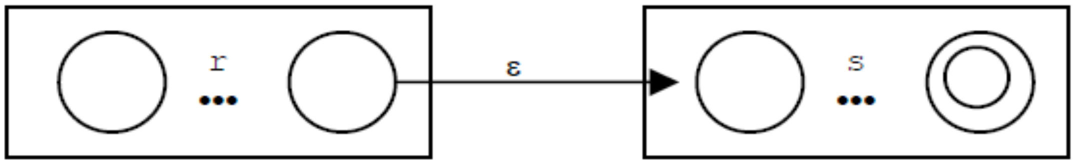


Basic Representation. If **r** is a regular expression, then the machine



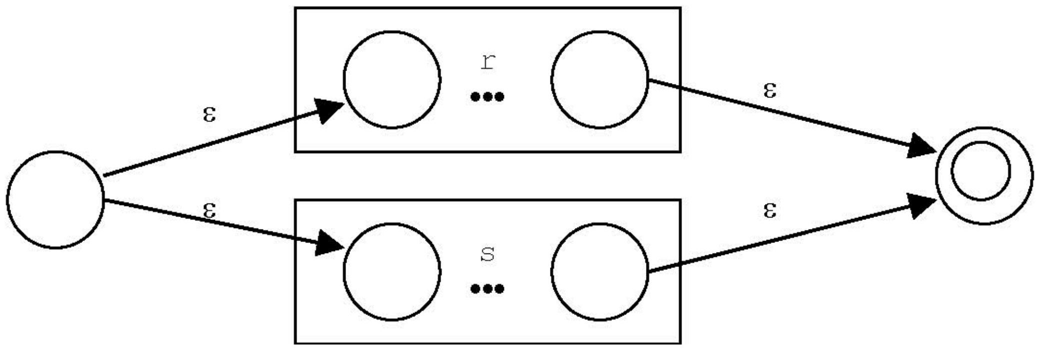
represents **r**.

Concatenation. If **r**and **s** are regular expressions, then



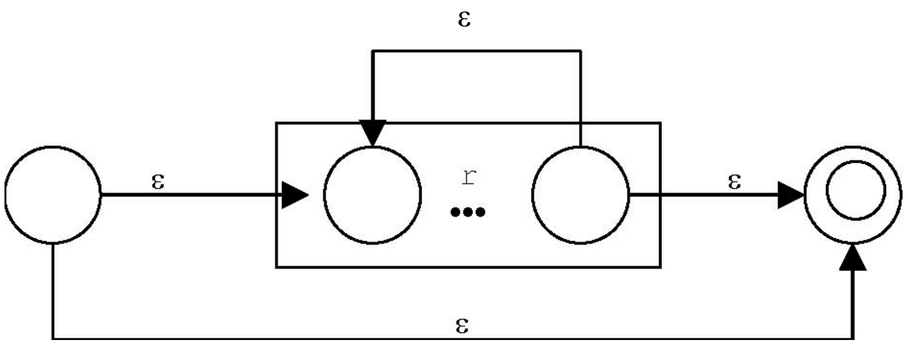
is the machine that accepts **rs.**

Choice. If **r**and **s**are regular expressions, then



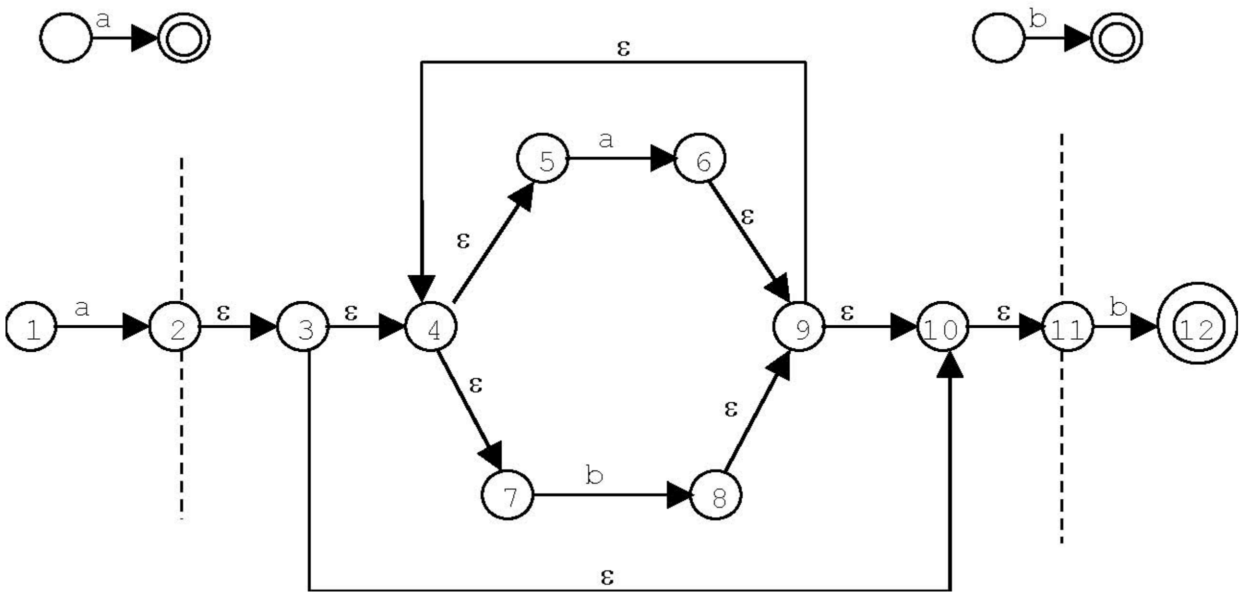
is the machine that accepts **r | s**.

Closure. If **r** is a regular expression than



is the machine that accepts **r***.

Now give the expression: **`a( a | b )* b,`**



Given an NFA, we can create a DFA using a technique called **Subset Construction**. The key idea in the subset construction is an ε-closure. Note, using $\underline{\mathbf{s}}$ instead of $\overline{\mathbf{s}}$

For a state **s**, the ε-closure $\underline{\mathbf{s}}$, is the set of all states, including **s** itself, that can be reached by 0 or more ε-transitions.

Given a set of states **S**, the ε-closure $\underline{\mathbf{S}} = \bigcup_{s \text{ in } S} \underline{\mathbf{s}}$

Given the previous NFA then:

$\underline{\mathbf{1}} = \{ 1 \}$ $\qquad$ $\underline{\mathbf{2}} = \{ 2, 3, 4, 5, 7, 10, 11\}$

$\underline{\mathbf{3}} = \{3, 4, 5, 7, 10, 11\}$ $\qquad$ $\underline{\mathbf{4}} = \{ 4, 5, 7\}$

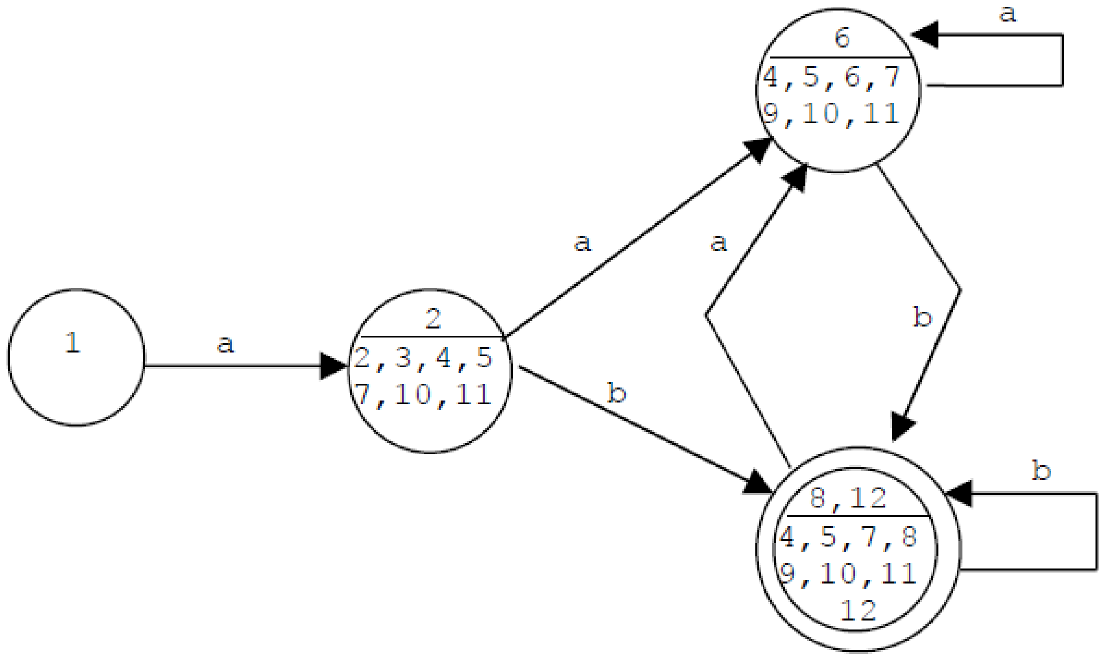
$\underline{\mathbf{5}} = \{5\}$ $\qquad$ $\underline{\mathbf{6}} = \{4, 5, 6, 7, 9, 10, 11\}$

$\underline{\mathbf{7}} = \{7\}$ $\qquad$ $\underline{\mathbf{8}} = \{4, 5, 7, 8, 9, 10, 11\}$

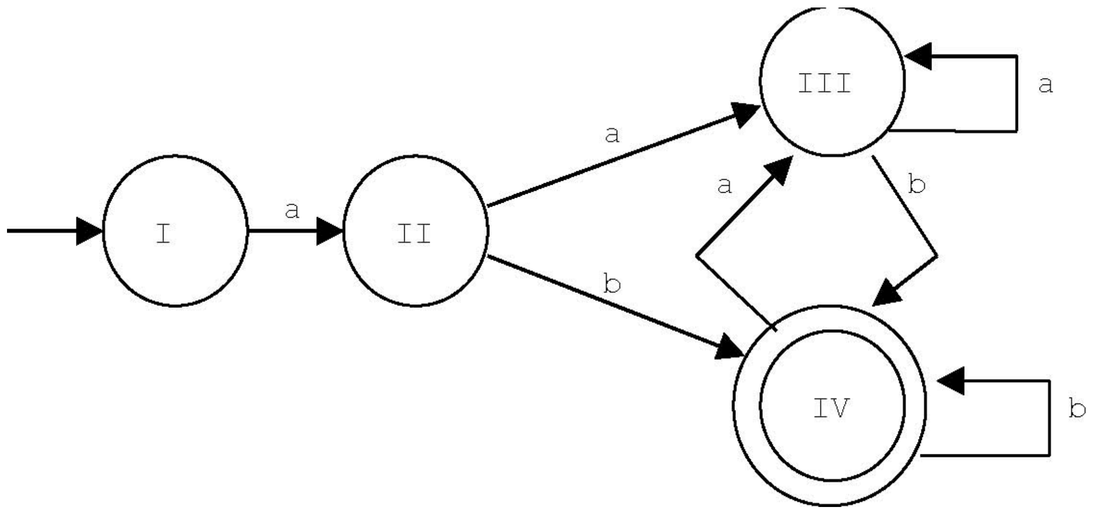
$\underline{\mathbf{9}} = \{4, 5, 7, 9, 10, 11\}$ $\qquad$ $\underline{\mathbf{10}} = \{10, 11\}$

$\underline{\mathbf{11}} = \{ 11 \}$ $\qquad$ $\underline{\mathbf{12}} = \{12 \}$ The accepting state

The subset construction gives us the following diagram



And, renaming the states gives the diagram



While this machine can accept any string for the regular expression **`a (a | b)* b`**, it has one extra state than the original machine. However, an examination shows that states II and III can be merged. This gives an equivalent machine to the first one presented.

Now it is time to consider an implementation of a scanner. There are two ways to get the input.

1 Read one line at a time. This is the how Louden's scanner works. This means that we can *pushback* a character into the input stream by simply decrementing an index.

```
import java.io.*;

/*
      This is an example of reading a TEXT file, one line at a time. The
      readLine() method returns a String. The returned String is null when
      the end of the file is reached.
*/

class LineReader {

   public static void main( String [] args ) throws IOException {
      File testFile = new File ( "test.data" );
      FileInputStream inStream = new FileInputStream( testFile );
      InputStreamReader inStreamReader = new InputStreamReader( inStream );
      BufferedReader reader = new BufferedReader( inStreamReader );

      String data = reader.readLine();
      while( data != null ) {
            System.out.println( data );
            data = reader.readLine();
      }
     reader.clos();
   }
 }
```

2. Read one character at a time. When using an object oriented language, we can use a separate class to read a file and to store the next character to be read in a class instance variable. Then instead of pushing an input character back into the stream, we can use a `peek()` method to look ahead. However, we also need a method to advance the input.

```
import java.io.*;

/*

      This is an example of reading a TEXT file one character at a time.The
      read() method returns an integer that must be cast to a character.When
      the end of the file is reached, the read() method returns a -1.

*/

class StreamReader {

   public static void main( String [] args ) throws IOException {
      File testFile = new File ( "test.data" );
      FileInputStream inStream = new FileInputStream( testFile );
      InputStreamReader reader = new InputStreamReader( inStream );

      int data = reader.read();
      while( data != -1 ) {
         char c = (char) data;
         System.out.print( c );
         data = reader.read();

      }
     reader.close();
   }
 }
```

Examine `scan.c` (Louden). Note how reserve words are handled.

Examining the Tiny Scanner

See Table 2.1 Reserved Words and Symbols. Note there is only one symbol that is two characters long and it is not a prefix.

See Figure 2.10: The DFA of the Scanner. Note that an `ID` is only letters (like C-) and a number is only digits (also like C-). This means that there are no decimal points, no exponential notation, or flags such as 31L.

Tokens are an enumerated type defined in **`globals.h`**. I am providing an interface called **`constants.java`** that creates static constants for use in the compiler.

The main function of the scanner is called **`getToken()`**, which returns a structure called **`TokenType`**. This is the enumerated type created in **`globals.h`**.

For our scanner, a token should be an object of a **`Token`** class, which—at a minimum—has two fields: one for the string value and one for the constant. Note that my **`Token`** class has four fields: **`type`**, **`line number`**, **`string value`**, and **`numeric value`**.

```
state = START;
while( ! done ) {
   c = getNextChar();
   save = true;
   switch( state ) {

       case START: //depending on the character c, change the state
                   //and turn save off

       case INID: //if c is not a letter, unread c and turn save off

       case INNUM:
         •
         •
         •

   } //end switch

   if( save ) // add character c to token string

   if( state == DONE )
        //if the current token is an ID, check if it is a reserved word

   }// end while loop

   return token;
```

An alternate way to having one large loop, reading a character at the top of the loop, and looking through each state (the switch statement), is to put while loops inside some of the case statements. For example, if the state is that for an identifier, place a while loop in the case statement to read the rest of the identifier, then to check for a reserved word and set the state to done.

Tokens

What data needs to be in a token? A token should be a separate class. It should contain the following fields:

- An **`int`** field to hold the constant identifying the token, such as: ELSE, ID, NUMBER.
- A **`string`** field to hold the lexeme of an identifier
- An **`int`** field to hold the value of a number
- An **`int`** field to hold the line number on which the token appeared.

Use Of A Class To Read A File

We can read a file either one line at a time (see `lineReader.java`) or one character at a time (see `streamReader.java`). In either case, you are advised to create a separate class to read the source file.

If reading a line at a time, the class should store the line as an instance variable or if reading a character at a time, the class should have a one character buffer that holds the "next character" to be read from the file. See the binary data file example.

While reading a line at a time can allow for the implementation of a "pushback" or an unread method, you are advised to implement the following three methods regardless of how you read the source file.

```
public char getNextChar()
   //This method returns the next character from the file and advances the
   //input.

public char peek()
   //This method returns the next character from the file but does not
   //advance the input.

public void advance()
   //This method does not return anything but advances the input
```

Two Character Symbols

One problem our scanner will need to deal with that the tiny scanner did not is two character symbols where the prefix character is also a valid symbol. For example, the symbols for *less than* and *less than or equal to* (**<** and **<=**) or the symbols for *is assigned* and *is equal to* (**=** and **==**).

Note that the Tiny scanner reads one line at a time. Since a **`string`** in C can be processed as an array (A **`string`** in C is just an array of **`char`**.), it is easy to get the next character (see the function **`getNextChar()`**), except when we try to put back the first character of a line.

Note that reserved words are stored in a table in the Tiny scanner, and, when an identifier is read, it is checked to see if it is a reserved word. You should also note the addition of **`read`** and **`write`** to the reserved words of C-.

In addition to reading the source file line by line, we can read a file character by character. In this case I recommend creating a separate class to deal with the file.

I made two methods: one that returns the next character and one that advances the next character but without returning it.

Rather than an **`ungetNextChar()`** method, I made a **`peek()`** method. You may discover that using a current character variable, which is instantiated at the beginning of the main loop with the `getNextChar()` method, and a next character variable, which is instantiated at the beginning of the main loop with the `peek()` method will be helpful is the two character symbol problem.

Note that we need to keep track of line numbers for error messages. I made a **`Disk`** class which deals with reading the source file. This class also keeps track of the line number. It increments the line number whenever it reads a new line character, and it has a method that returns the current line number.

My scanner also makes use of some of the methods in the **`Character`** class such as: **`isDigit()`**, **`isLetter()`**, and **`isWhiteSpace()`**.

Chapter 3: *Context-Free Grammars and Parsing*

The concept of parsing is to consider or determine the **syntax** of a program. The syntax of a language is normally given by a **grammar**.

The **Chomsky Hierarchy** gives us several levels of grammars. At the lowest level are the regular languages, which are defined by regular expressions (grammars). However, regular expressions do not allow for nested structures and so are not powerful enough to define a programming language.

The next level up in the Chomsky Hierarchy is **Context-Free** languages. Almost all programming languages fall into this category.

A context-free language is represented by a set of productions of the form

$A \rightarrow X_1 X_2 X_3 \ldots X_n$, where A is a single symbol. An X_i, which may be ε, is called a **non-terminal** if it appears on the left side of a production, and a **terminal** if it appears only on the right side of a production.

The idea of a production is that whenever an A appears in the right side of a string, it may be replaced by the string $X_1 X_2 X_3 \ldots X_n$.

There is a distinguished symbol—often called S in abstract grammars—that is known as the **start** symbol. A string of non-terminals is a legal string in the language defined by the grammar if there is a series of productions that starts from the start symbol and ends with the string of non-terminals.

Parsing is the process of verifying that a string of non-terminals can be derived from the start symbol.

The grammar rules of a language are normally expressed in a form called BNF (**Backus-Naur Form** or sometimes Backus Normal Form). John Backus was the first to use this form when he presented the grammar for Algol60. Peter Naur edited the Algol60 report and adapted the format.

BNF uses the metacharacter | for choice. We will see a version called EBNF or extended BNF that uses several other metacharacters.

An example of a BNF grammar is

```
Exp → Exp Op Exp
Exp → ( Exp )
Exp → number
Op → +
Op → -
Op → *
```

Where number is defined as

```
digit = 0 | 1 | ... | 9
number = digit digit*
```

If I want to show that the string `(5 - 2) * 4` is a legal string in this grammar, there are two ways I could proceed.

```
Exp  ⇒  Exp Op Exp
     ⇒  ( Exp ) Op Exp
     ⇒  ( Exp Op Exp ) Op Exp
     ⇒  ( number Op Exp ) Op Exp
     ⇒  ( 5 Op Exp ) Op Exp
     ⇒  ( 5 - Exp ) Op Exp
     ⇒  ( 5 - number ) Op Exp
     ⇒  ( 5 - 2 ) Op Exp
     ⇒  ( 5 - 2 ) * Exp
     ⇒  ( 5 - 2 ) * number
     ⇒  ( 5 - 2 ) * 4
```

This is both a **top-down** derivation and a **left-most** derivation. Alternately, I could start from `(5 - 2) * 4` and reverse the steps; that is, finding the right side of production in the string and replacing the right side with the left side. This is called a **bottom-up** derivation.

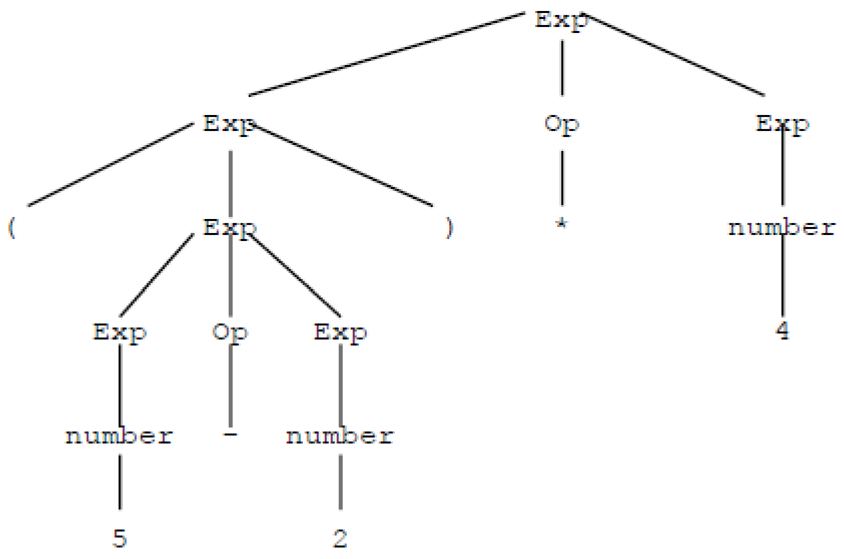
The terms top-down and bottom up refer to another way to visualize the derivation process by using a tree structure called a **parse tree**.

The root of the parse tree is the start symbol. An internal node of the tree is a non-terminal and the children of the node are given by the right side of a production involving the parent. The leaves of the tree are terminals and form the string being parsed.

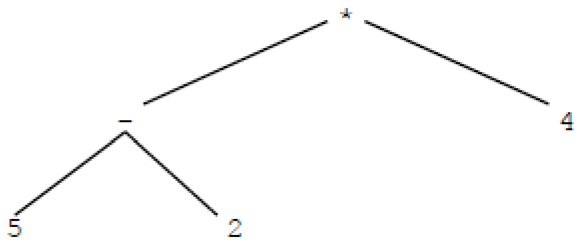
A parse tree usually contains more information than is actually needed to display the structure of a program. So a program is usually represented by a simplified tree called an **Abstract Syntax Tree**.

For example:

Parse Tree



Abstract Syntax Tree



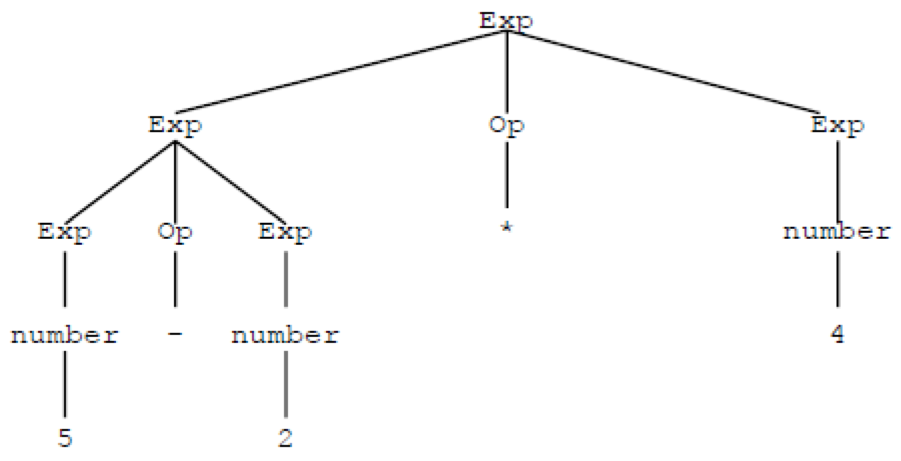
The AST contains the essence of the program.

Thus, the role of the parser is to take tokens, one-at-a-time, from the scanner and to verify that they form syntactically correct string according to the rules of the grammar and to produce an abstract representation of the program. This abstract representation is usually an AST.

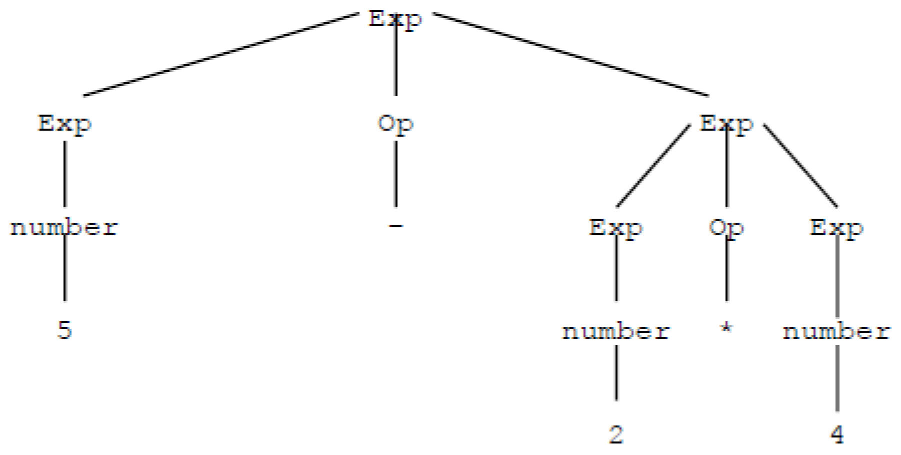
Ambiguous Grammars

It is possible for a grammar to allow a string to have more than one parse tree. For example, the string `5 - 2 * 4` could be represented by the two trees

This is a leftmost derivation



or, the string could be represented by a rightmost derivation.



A grammar that is ambiguous is a problem for a parser. One way to deal with ambiguity is to provide **disambiguating rules**. Another method is to alter the grammar so that it produces only one parse tree.

Precedence

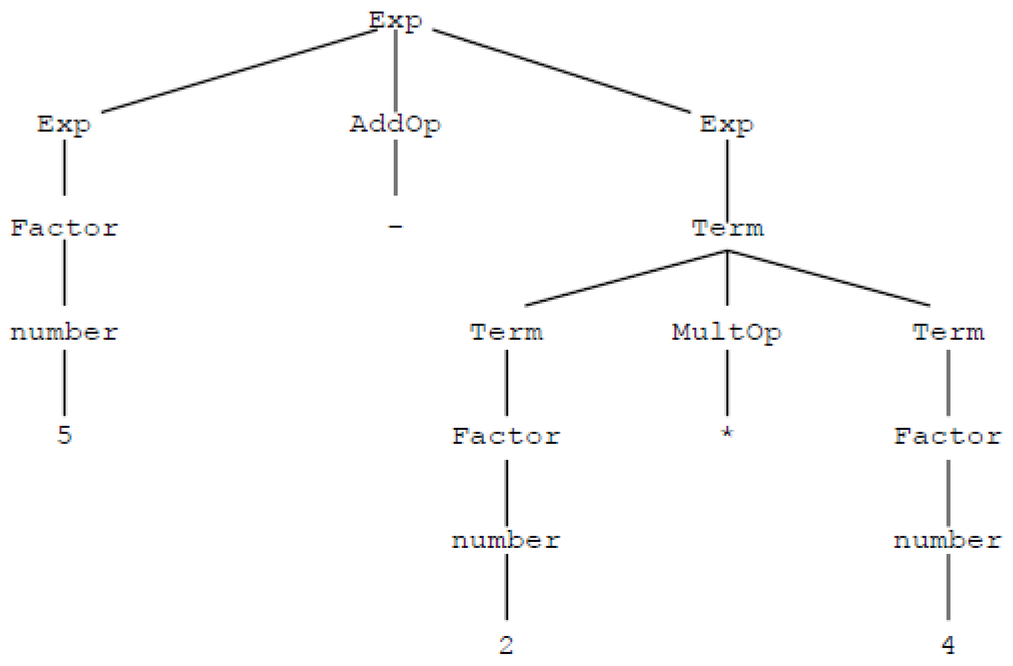
A problem with the previous example is that the grammar does not specify that multiplication and division have precedence over addition and subtraction.

To show that multiplication and division have higher precedence, we want to insure that expressions involving these operations in an unparenthesized expression are **lower** in the parse tree than expressions involving addition and subtraction. That is, the second parse tree, the rightmost derivation, is the correct derivation.

The grammar to fix this problem is

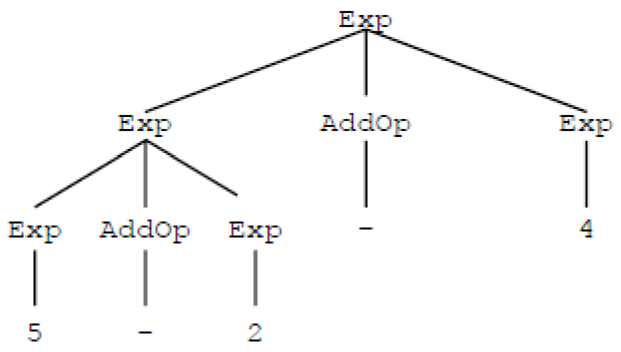
```
Exp     → Exp AddOp Exp | Term

AddOp   → + | -

Term    → Term MultOp Term | Factor

MultOp  → *

Factor  → ( Exp ) | number
```

Thus, the string `5 - 2 * 4` has only one parse tree

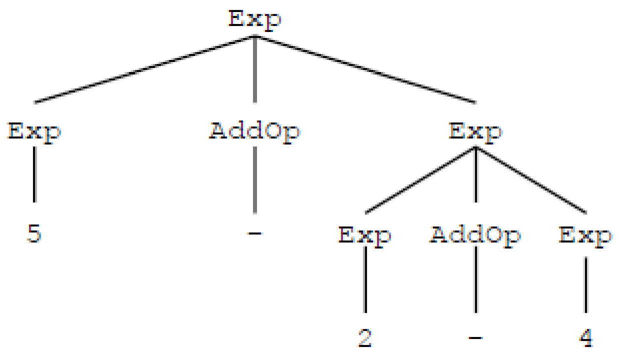


Associativity

Note that the string `5 - 2 - 4` still has two parse trees



Or



The reason there is no default associativity is that the production: `Exp → Exp AddOp Exp` is both **right recursive** and **left recursive**. That is, a right recursive rule makes its operators right associative, and a left recursive rule makes its operators left associative.

So, the final BNF for this grammar is

```
Exp → Exp AddOp Term | Term

AddOp → + | -

Term → Term MultOp Factor | Factor

MultOp → *

Factor → ( Exp ) | number
```

The Dangling Else Problem

Given: `if( X ) if( Y ) S1 else S2`   Is it

```
if( X )
   if( Y ) S1
else S2
```

or is it

```
if( X )
   if( Y ) S1 else S2
```

This can be fixed in one of three ways.

1 Change the grammar: see page 121.

2 Bracketing keywords; i.e. `end if` or `fi`.

3 Disambiguating rule. This is how it is most often done.

EBNF

Extended BNF uses metacharacters to make writing expressions easier. It does not express anything that could not be expressed using only BNF.

Repetition

With BNF notation, repetition is expressed using recursion.

```
S → S b | a
```

This is an "a" followed by zero or more "b"s. For example: `a, abb, abbb,` …. This can be expressed in EBNF as `a{b}` . In some books you will see `ab*` . This is what I used for the EBNF for C minus.

Instead of writing
```
   Exp → Exp AddOp Term | Term
```
we can write
```
   Exp → term { AddOp Term }
```

The general form for
```
   N → N Y | X
```
is
```
   N → X { Y }
```

Choice

Optional Components in EBNF are shown using the [ ] construct. For example
```
   if-statement → if ( Exp ) statement [ else statement ]
```

A second way to express EBNF is by a graphical means called a **syntax diagram**, sometimes called a rail diagram.

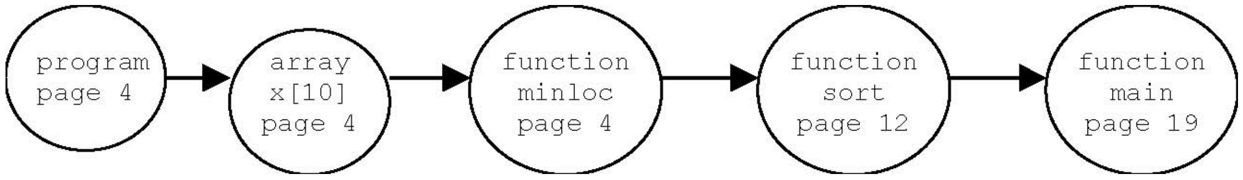
Syntax Tree

The AST for this project is patterned after the syntax tree for the Tiny language. See pages 135 to 138 of the textbook. It is made of objects from the `TreeNode` class. Note that the specification for the `TreeNode` class is in the Abstract Syntax Tree handout. See pages 19 and 20 for examples of the `read` and `write` statements.

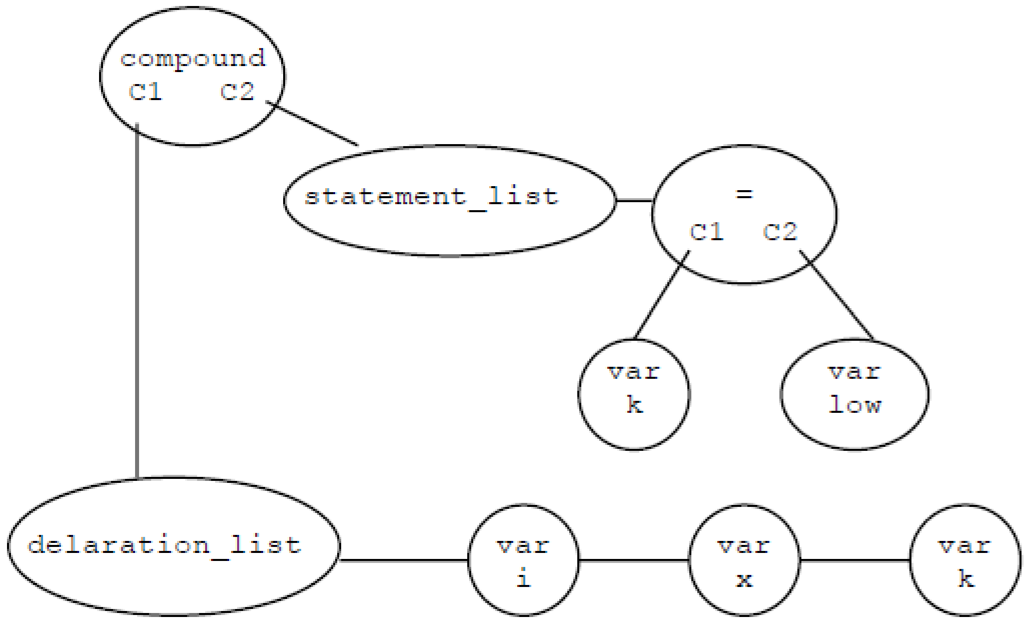
Each node has several data fields. Most of the information for these fields come from the token that causes the `TreeNode` instance to be created.

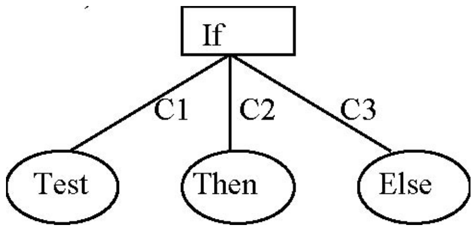
The nodes of the AST contain 4 pointers: three child pointers and one sibling pointer. The sibling pointer is for structures that are linked lists. For example, a `statement_list` is a sequence of various types of statements. How many statements? We don't know ahead of time; so, these nodes are linked by sibling pointers.

Refer to the handout:



Now consider pages 5 and 6 of the handout.

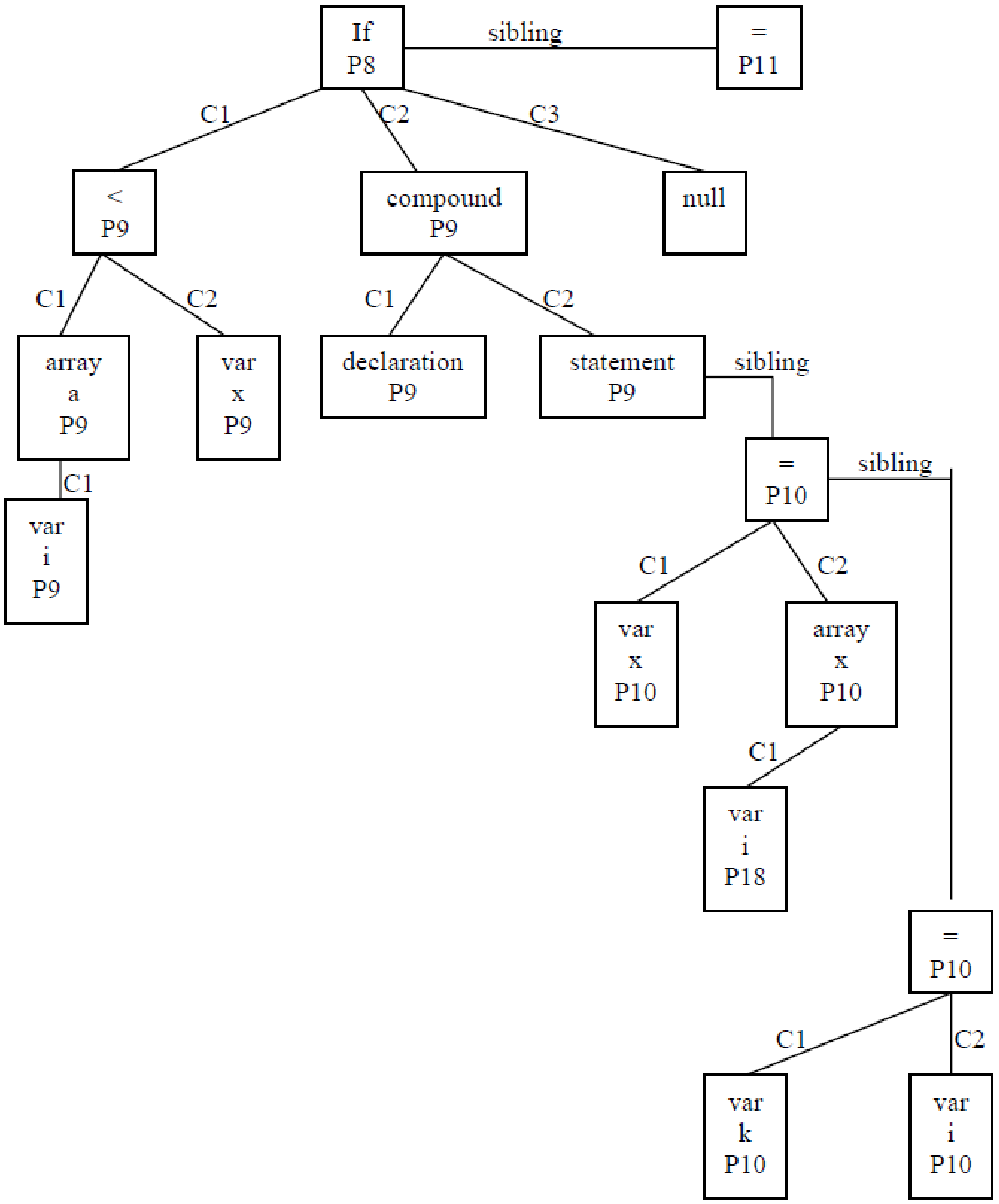




Additional Examples

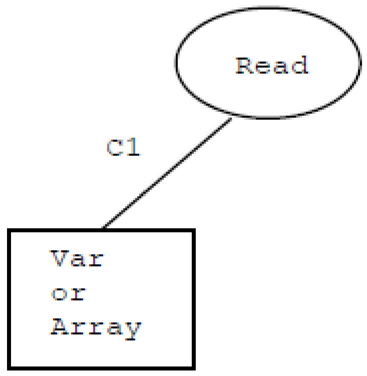
Consider the following (bottom of page 8 of the handout.

```
if ( a[ i ] < x ) {
   x = a[ i ];
   k = i;
}
```

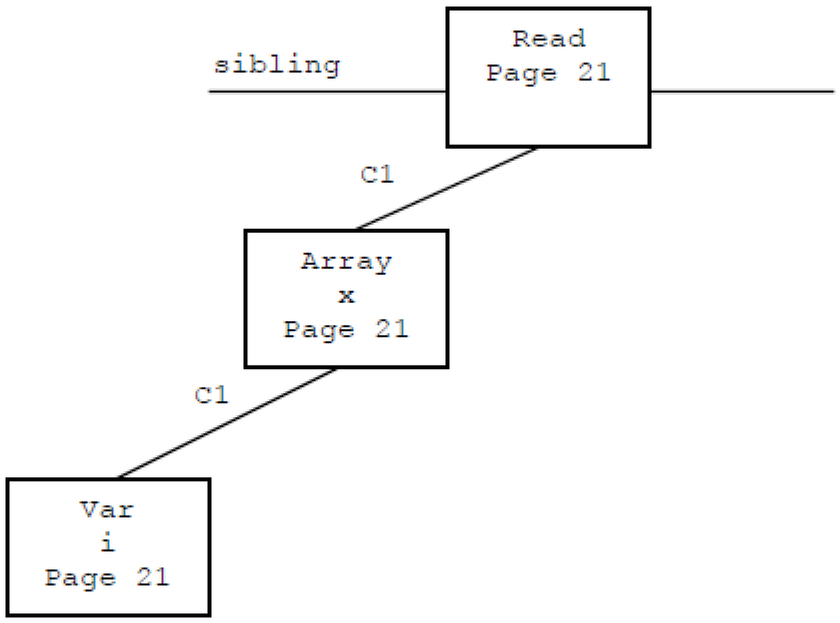

If
P8
sibling
=
P11
C1
C2
C3
<
P9
compound
P9
null
C1
C2
C1
C2
array
a
P9
var
x
P9
declaration
P9
statement
P9
sibling
C1
var
i
P9
=
P10
sibling
C1
C2
var
x
P10
array
x
P10
C1
var
i
P18
=
P10
C1
C2
var
k
P10
var
i
P10

Additional Examples of the Abstract Syntax Tree using the read and write statements.

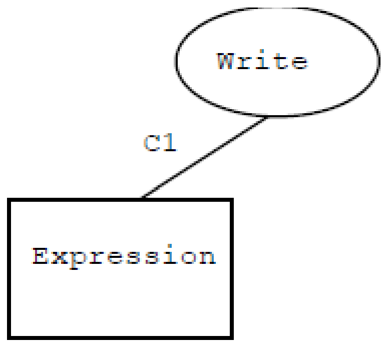
1. `Read`: see page 21 of syntax tree example



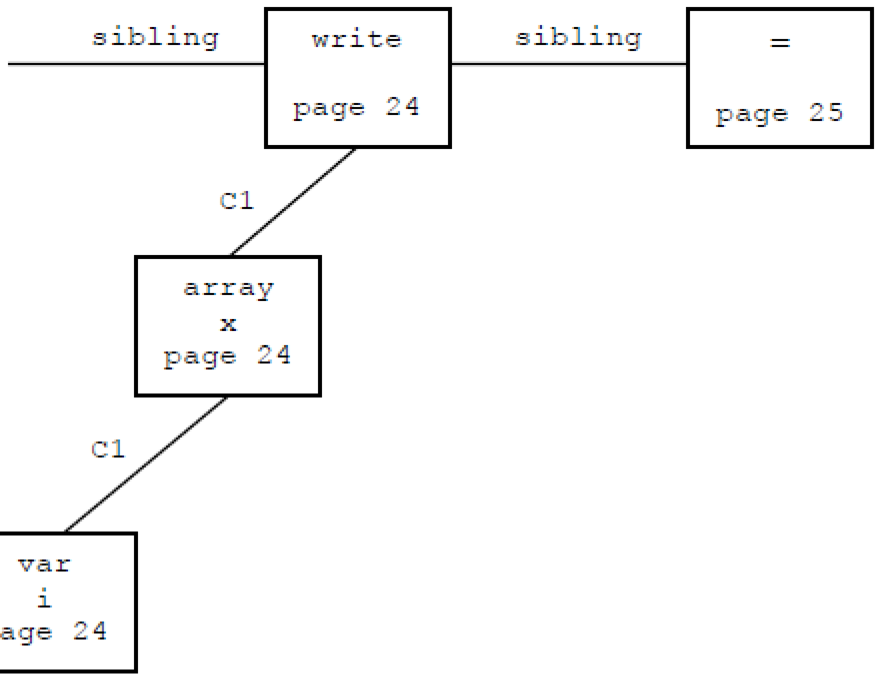
2. See page 21 of syntax tree example. `read x` [i];



3. `Write`: See page 24



4. See page 24 of syntax tree example: `write ( x[ i ] );`



Chapter 4: *Top-Down Parsing*

Parsing algorithms come in two general varieties: **top-down** or **bottom-up**.

In as sense, a top-down parser checks that a string is legal by starting with the start symbol and using a leftmost derivation to end up with the string in question.

Top-down parsing algorithms are either **backtracking**, which is not practical, or **predictive**. The two predictive algorithms in this chapter are called **Recursive Descent** and **LL(1)**.

Recursive descent is used most often when writing a compiler by hand. So, it is the algorithm we will use for our project**.**

The other algorithm, LL(1) means that the input is processed from left to right and a leftmost derivation with 1 lookahead symbol is used. LL(1) parsers are not used anymore since the automated tools such as YACC produce bottom-up parsers.

Section 4.1 Recursive Descent Parsing

The basic idea of a recursive descent parser is straight forward. Each non-terminal in a grammar forms a method, where the right side of the production determines the body of the method.

Assume there is a class variable called `currentToken` . Also assume there is a class variable called `currentType`, which is obtained from `currentToken`.

Now assume there is a method called `match()` that takes a token type as a parameter. It compares the parameter to `currentType`. If they do not match, it returns an error.

```
token currentToken;
int currentType, currentLineNumber;


private void match( int expectedType ) {
   if( currentType == expectedType) {
      currentToken = Scanner.getToken();
      currentType = currentToken.getType();
      currentLineNumber = currentToken.getLineNumber();
   }
   else
     //print appropriate error message and crash
```

You might find it useful to have a similar method called `accept()` that performs the first part of `match()` but without doing the check.

Consider a read statement. The EBNF for a read statement is

```
Read_Statement → read variable ;
```

This become a method such as the following.

```
private void read_statement() {
   match( Constants.READ );
   variable();
   match( Constants.SEMI );
}
```

However, the parser should be building a syntax tree while it parses the program. Therefore, the method should be more like the following

```
private TreeNode read_statement() {
   TreeNode t = new TreeNode();
   t.lineNumber = currentLineNumber;

   match( Constants.READ );
   t.nodeType = Constants.READ;

   t.C1 = variable();
   match( Constants.SEMI );
   return t;
}
```

In order to develop a recursive descent parser, we must first transform a grammar expressed in BNF to EBNF. This is because we **must** transform any left recursion to repetition. Why would left recursion be bad in a recursive descent parser?

Consider statement 22 from the BNF grammar of C- (page 492).

```
additive-expression → additive-expression addOp term | term
```

This needs to have the recursion removed. Recall the rule

```
N → N Y | X ⇒ N → X { Y }
```

Now if `N` is `additive-expression`, `Y` is `addOp term`, and `X` is `term`, the statement becomes

```
additive-expression → term { addOp term }
```

See #23 of the EBNF grammar. Using this grammar rule, the method for additive-expression becomes   (Use **`x + y - z`** as an example)

```
private TreeNode additiveExpression() {
   TreeNode t, tmp;

   t = term();

   while( currentType == Constants.PLUS || currentType == Constants.MINUS) {
      tmp = new TreeNode();
      tmp.nodeType = currentType;tmp.lineNumber = currentLineNumber;
      accept();

      tmp.C1 = t;
      t = tmp;
      t.C2 = term();
   }

   return t;
 }
```

Section 4.2 LL(1) Parsing

LL(1) parsing uses an explicit stack rather than recursive calls for parsing. It also uses a table to control the parsing action. Consider the following grammar

```
Exp → Term Exp′

Exp′ → addOp Term Exp′ | ε

addOp → + | -

Term → Factor Term′

Term′ → multOp Factor Term′ | ε

multOp → *

Factor → ( Exp ) | number
```

Now consider the Table 4.4 on page 163. The rows are labeled by non-terminals and the columns are labeled by terminals.

We initialize the parsing stack to `$ S` where '`$`' indicates an empty stack and `S` is the start symbol.

The input is written left to right with a '`$`' on the right side indicating the end of the input. If the top of the stack is `$` and the input is `$`, then we accept the input as legal.

The basic algorithm is (1) replace a non-terminal `A` at the top of the stack with a string α using a grammar rule `A` → α . (2) Match the terminal at the top of the stack with the next input token.

As an example, consider parsing `3 + 5`

Stack	Input	Action
`$ Exp`	`3 + 5 $`	`Initialize`
`$ Exp′ Term`	`3 + 5 $`	`Exp → Term Exp′`
`$ Exp′ Term′ Factor`	`3 + 5 $`	`Term → Factor Term′`
`$ Exp′ Term′ number`	`3 + 5 $`	`Factor → number`
`$ Exp′ Term′`	`+ 5 $`	`Match, number = 3`
`$ Exp′`	`+ 5 $`	`Term′ → ε`
`$ Exp′ Term addOp`	`+ 5 $`	`Exp′ → addOp Term Exp′`
`$ Exp′ Term +`	`+ 5 $`	`addOp → +`
`$ Exp′ Term`	`5 $`	`Match +`
`$ Exp′ Term′ Factor`	`5 $`	`Term → Factor Term′`
`$ Exp′ Term′ number`	`5 $`	`Factor → number`
`$ Exp′ Term′`	`$`	`Match, number = 5`
`$ Exp′`	`$`	`Term′ → ε`
`$`	`$`	`Exp′ → ε`
`$`	`$`	`Accept`

Note how important it is to be able to create the table.

Ambiguous Grammars

A grammar is not LL(1) if there is more than one production choice in a table. Part of what causes the problem is when there is repetition and choice in the grammar. Two techniques that are used to rewrite the BNF are **left-factoring** and **removal of left-recursion**.

Left-Recursion Removal

Given $A \rightarrow A\alpha \mid \beta$, where α and β are strings and β does not begin with `A`. Then

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

For example: $\overset{A}{\texttt{Exp}} \rightarrow \overset{A}{\texttt{Exp}}\ \overset{\alpha}{\texttt{addOp Term}} \mid \overset{\beta}{\texttt{Term}}$ becomes

`Exp → Term Exp′`

`Exp′ → addOp Term Exp′ | ε`

Left-Factoring

We need to use left-factoring when 2 or more strings begin with a common sequence.

$A \rightarrow \alpha\beta \mid \alpha\gamma$ becomes

```
A  → α A′
A′ → β | γ
```

For example:

```
if-statement → if ( exp ) statement |
               if ( exp ) statement else statement

α = if ( exp ) statement
β = ε
γ = else statement

if-statement → if ( exp ) statement if-statement′
if-statement′ → else statement | ε
```

First and Follow sets

To complete the algorithm to construct the LL(1) table, we need to construct the **first** and **follow** sets.

First Sets

If α is any string of grammar symbols, then First(α) is the set of terminals that begin strings derived from α. If $\alpha \Rightarrow^* \varepsilon$, then ε belongs to First(α).

To compute First(X),

1. If X is a terminal, then First(X) = { X }.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to First(X).
3. If X is a non-terminal and $X \rightarrow Y_1\ Y_2 \ldots Y_n$ is a production, then everything in First(Y_1) is in First(X). If Y_1 does not produce ε, then we are done. However, if $Y_1 \Rightarrow^* \varepsilon$, then add First(Y_2) to First(X) and so on.

Now to compute First($X_1\ X_2 \ldots X_n$) for any string, add all non-ε symbols in First(X_1) to First($X_1\ X_2 \ldots X_n$). If First(X_1) contains ε, add all non-ε symbols of First(X_2) to First($X_1\ X_2 \ldots X_n$) and so on. If ε belongs to First(X_i) for all i, then add ε to First($X_1\ X_2 \ldots X_n$).

Definition. A non-terminal A is called **nullable** if there is a sequence of productions such that $A \Rightarrow^* \varepsilon$. A is nullable iff First(A) contains ε.

Follow Sets

For a non-terminal `A`, the set Follow(`A`) is the set of terminals that can appear immediately to the right of `A` in some **sentential** form. That is, Follow(`A`) is the set of terminals, `a`, such that there exists a derivation of the form $S \Rightarrow^* \alpha\ A\ a\ \beta$. If `A` can be the rightmost symbol in a sentential form, then `$` is in Follow(`A`).

To compute Follow(`A`) for any non-terminal `A`, apply the following rules until there are no changes.

1 Place `$` in Follow(`S`) where `S` is the start symbol and `$` is the end of input marker.

2 If there is a production $A \rightarrow \alpha\ B\ \beta$, then everything in First(β) except for ε is in Follow(`B`).

3 If there is a production $A \rightarrow \alpha\ B$, or a production $A \rightarrow \alpha\ B\ \beta$, where First(β) contains ε ($\beta \Rightarrow^* \varepsilon$), then everything in Follow(`A`) is in Follow(`B`).

Example: Constructing the First and Follow sets

```
Decl → Type Var-List

Type → int | float

Var-List → ID , Var-List | ID
```

To begin, this must be left-factored.

```
Decl → Type Var-List

Type → int | float

Var-List → ID Var-List′

Var-List′ → , Var-List | ε
```

Terminals
First(`int`) = { `int`}
First(`float`) = { `float`}
First(`,`) = { `,`}
First(`ID`) = { `ID`}

Non-Terminals

First(`Decl`) = { `int`, `float`}	Follow(`Decl`) = { `$`}
First(`Type`) = { `int`, `float`}	Follow(`Type`) = { `ID`}
First(`Var-List`) = { `ID`}	Follow(`Var-List`) = { `$`}
First(`Var-List'`) = { `,`, ε }	Follow(`Var-List'`) = { `$`}

Constructing the LL(1) Parse Table

A grammar in BNF is LL(1) if

1 For every production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ First(α_i) ∩ First(α_j) is empty for all $1 \leq i < j \leq n$.

2 For every non-terminal `A` such that First(`A`) contains ε, First(`A`) ∩ Follow(`A`) is empty.

If a grammar is LL(1), we can construct the LL(1) parse table as follows. Let the parse table be called M. M has a row for each non-terminal and a column for each terminal. For each non-terminal `A` and production choice $A \rightarrow \alpha$, do the following.

1. For each token `a` in First(α), add $A \rightarrow \alpha$ to M[`A`, a].

2. If ε is in First(α), for each element, `a`, of Follow(`A`), either a token or `$`, add $A \rightarrow \alpha$ to M[`A`, `a`].

Example Construct the LL(1) parse table of the previous grammar.

R1. `Decl → Type Var-List`
R2. `Type → int`
R3. `Type → float`
R4. `Var-List → ID Var-List'`
R5. `Var-List' → , Var-List`
R6. `Var-List' →` ε

	`Int`	`Float`	`ID`	`,`	`$`
`Decl`	R1	R1			
`Type`	R2	R3			
`Var-List`			R4		
`Var-List'`				R5	R6

Examples: Show the parse of `int x, y` and show that `int x, float y` is not legal.

Error Recovery in Top-Down Parsers

A parser that only determines if a program is syntactically correct or not is called a **recognizer**. At a minimum, a parser should be a recognizer, but it should also provide some indication of what error has occurred.

Sometimes a parser may attempt error correction, but this can be quite difficult.

Most error recovery techniques are ad-hoc, but there are some basic principles.

1 A parser should try to determine that an error has occurred as soon as possible.

2 After an error has occurred, the parser must pick a likely place to resume the parse. It should attempt to resume the parse as close to the original error as possible.

3 A parser should try to avoid the **error cascade problem** in which one error generates a long sequence of spurious error messages. (COBOL story)

4 A parser must avoid infinite loops on error in which an unending cascade of error messages is generated without consuming any input.

Panic Mode

Panic mode is a standard form of error recovery for recursive descent parsers. The basic idea is that when an error is encountered, the parser consumes tokens until it finds a place to restart the parse. The way this is done is to pass in as a parameter to each recursive call a set of tokens called a **synchronizing set**. When an error is encountered, the parser consumes tokens until a token in the synchronizing set is seen. The parse then continues from this point.

One logical candidate for the synchronizing set is the Follow set. Sometimes the First set is also used so that major constructs are not skipped.

Error Recovery in LL(1) Parsers

In an LL(1) parser, there are 3 basic alternatives for error recovery.

1 Pop the current symbol from the stack.

2 Advance the input.

3 Push a new symbol on the stack

Generally option 3 is a very ad hoc method and is rarely used. For example, the stack is popped, leaving it empty but with input still to be processed. One possibility is to push the start symbol onto the stack and advance the input until an element in the First set of the start symbol is seen.

So, with the other 2 options, generally if the input token is `$` or is in the Follow set of the current symbol on the stack, we pop the stack. Otherwise; advance the input.

Chapter 5: *Bottom-Up Parsing*

Bottom-up parsing uses an explicit stack. At the beginning of a parse, the stack is empty, and a successful parse ends with the input empty and the start symbol at the top of the stack.

A bottom-up parser is sometimes called a **shift-reduce** parser because there are two basic actions that we can take.

A **shift** means to push the current input token on the stack.

A **reduction** means that there is a production `A` → α and that α has been recognized as being on the top of the stack. In a reduce operation, α is popped and A is pushed on the stack.

Example: Parse `x + x * x` given the following grammar

```
E → E + T | T
T → T * F | F
F → ( E ) | x
```

Stack	Input	Action
	`x + x * x`	`Initialize`
`x`	`+ x * x`	`Shift`
`F`	`+ x * x`	`F → x`
`T`	`+ x * x`	`T → F`
`E`	`+ x * x`	`E → T`
`E +`	`x * x`	`Shift`
`E + x`	`* x`	`Shift`
`E + F`	`* x`	`F → x`
`E + T`	`* x`	`T → F`
`E + T *`	`x`	`Shift`
`E + T * x`		`Shift`
`E + T * F`		`F → x`
`E + T`		`T → T * F`
`E (start symbol)`		`E → E + T`

Thus, in general, a shift-reduce parser shifts input tokens onto the stack until it recognizes the right side of a production—sometimes called a **handle** in an abuse of notation, then reducing the handle.

The key to this style of parsing is to know when to shift and when to reduce and which production to use for the reduction. This is usually controlled by a table.

There are several different bottom up parsing algorithms which vary in complexity and power. We will start with the simplest algorithm: LR(0).

LR(0) Parsing

The 'L' in LR(0) means that the input is processed left to right; the 'R' indicates a bottom up parse; and the '0' indicates that no lookahead symbols are used.

An **LR(0) item** of a context-free grammar is a production with a distinguished position in its right-hand side. The position is normally indicated by a dot.

In addition, the grammar is augmented by a new start symbol. For example, the grammar

```
   L → ( L ) | a
```

becomes

```
   L′ → L
   L → ( L ) | a
```

Here, `L′` is the new start symbol. To construct an LR(0) parse table, we first construct the LR(0) items.

```
1.  L′ →• L
2.  L′ → L •
3.  L →• ( L )
4.  L → ( • L )
5.  L → ( L • )
6.  L → ( L ) •
7.  L →• a
8.  L → a •
```

In general, if `A` $\rightarrow \alpha$, where $\alpha = \beta\,\gamma$,

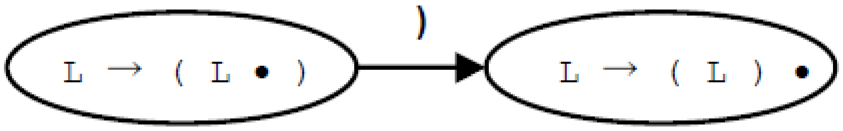
`A` $\rightarrow \bullet\ \alpha$ (called an **initial item**) means that we are about to recognize `A` $\rightarrow \alpha$.

`A` $\rightarrow \beta \bullet \gamma$ means that β is already on the stack and we are expecting to recognize γ.

`A` $\rightarrow \alpha\ \bullet$ (called a **completed item**) means that α now resides on the top of the stack and may be replaced with `A`.

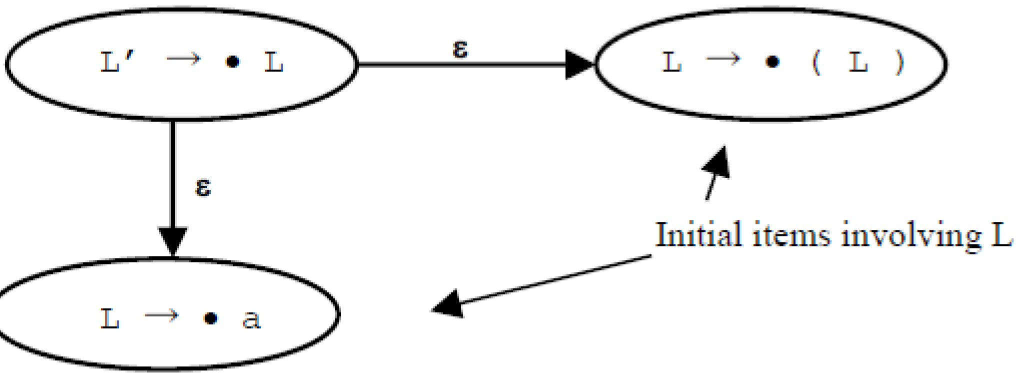
The next step is to create an NFA where a state contains an item.

If a state contains $A \rightarrow \alpha \bullet x\ \beta$ and x is a terminal, then there is a transition from the state $A \rightarrow \alpha \bullet x\ \beta$ to the state $A \rightarrow \alpha\ x \bullet \beta$ and the arc is labeled with x. So, for example, in the NFA for the example above, there is a transition

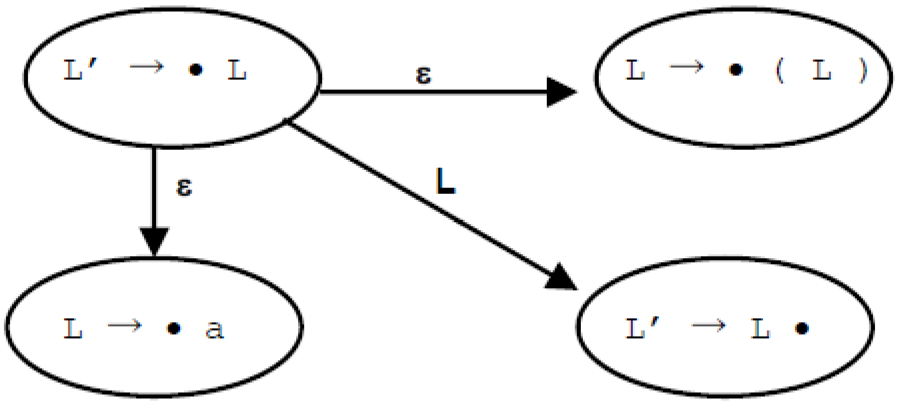


This transition represents a shift action.

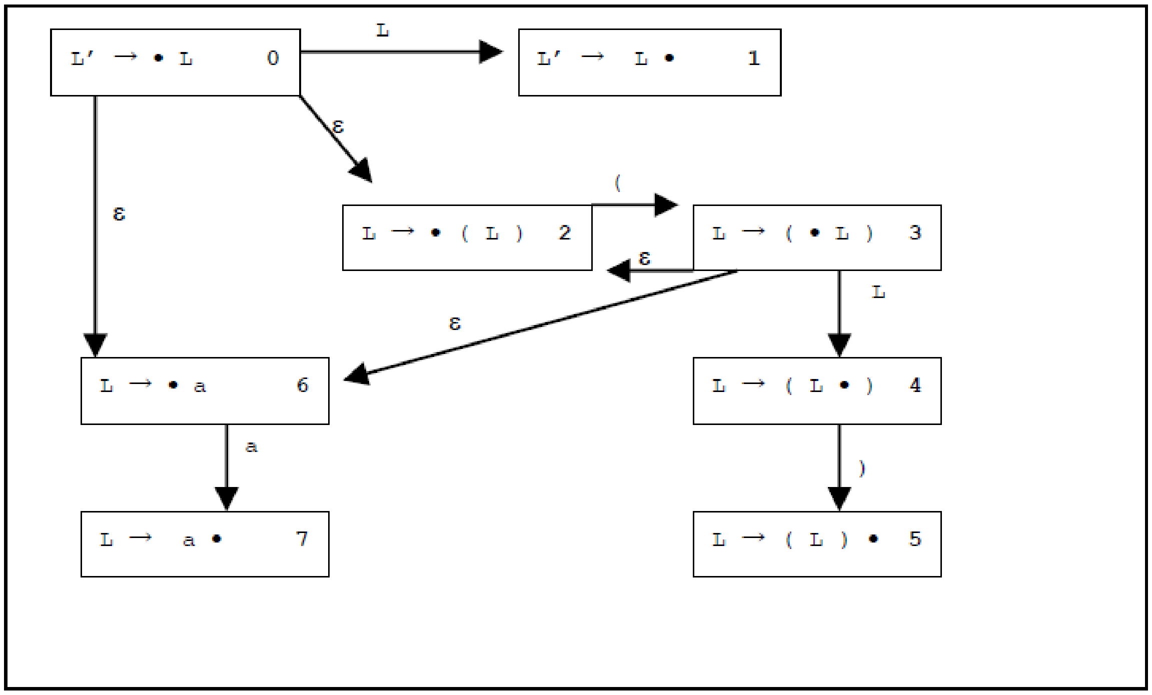
On the other hand, for a transition $A \rightarrow \alpha \bullet X\ \beta$, where X is a non-terminal, the process is more complex. Since X never actually appears in the input as a token, the only way that X can get shifted onto the parsing stack is when a production $X \rightarrow \gamma$ is parsed. So, in the NFA, we use an ε-transition to all initial items involving X. Thus:



However, we must also include a transition on X to tell us what to do if X is recognized. Thus the previous NFA has the following states.



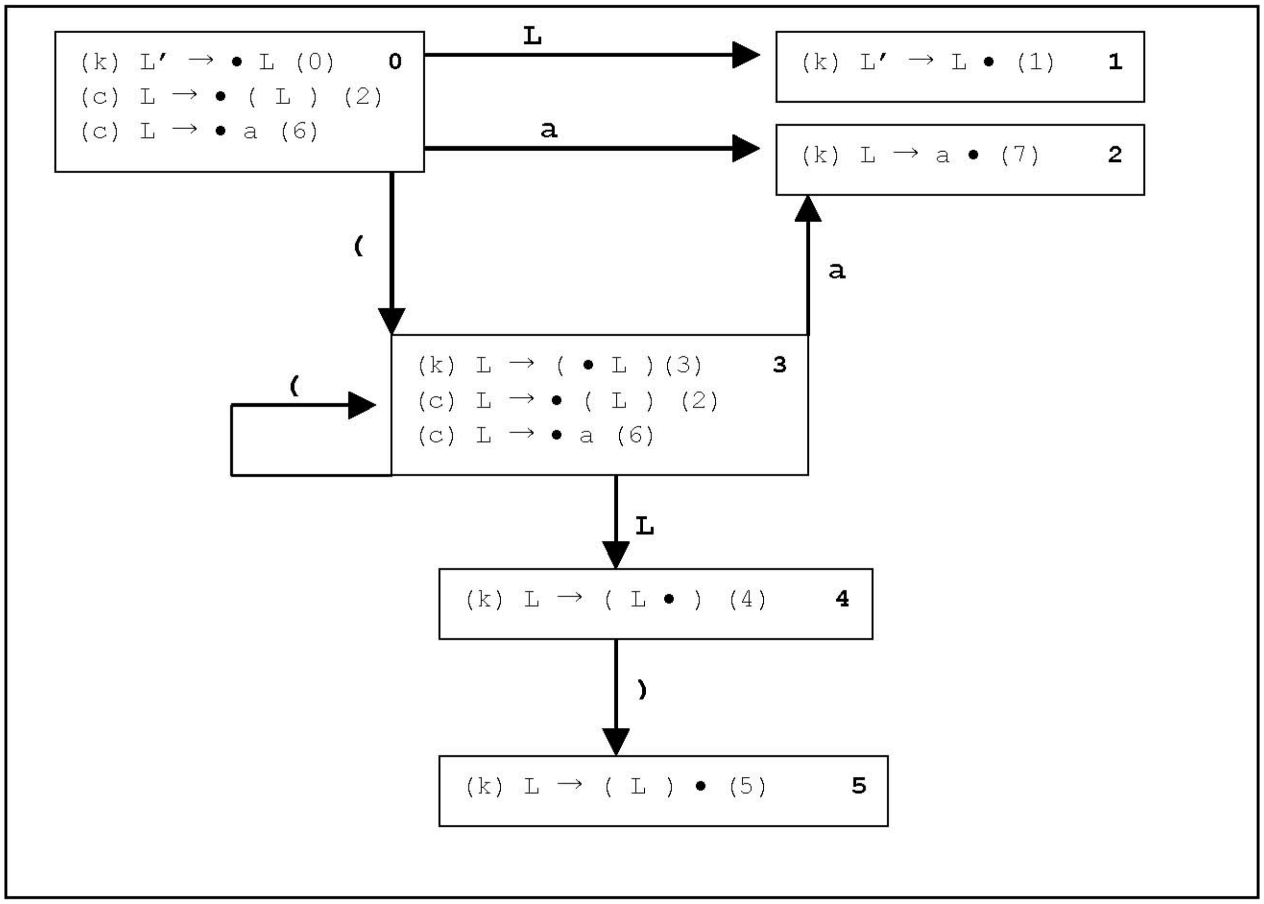
Therefore, the complete NFA for this grammar is



The next step is to use the subset construction technique to create a DFA.

<u>0</u> = { 0, 2, 6 }	**<u>1</u>** = { 1 }	**<u>2</u>** = { 2 }	**<u>3</u>** = { 2, 3, 6 }
<u>4</u> = { 4 }	**<u>5</u>** = { 5 }	**<u>6</u>** = { 6 }	**<u>7</u>** = { 7 }

Using the closure set above, the DFA is the following



In the above diagram, some items are labeled with a 'k' and some with a 'c'. The 'c' stands for a **closure** item: one that is part of a state because of an ε transition. The 'k' refers to a **kernel** item: one that is part of a state because of a non-ε transition.

Now consider the action for a parse of the expression `( ( a ) )`

Stack	Input	Action
`$ 0`	`( ( a ) ) $`	
`$ 0 ( 3`	`( a ) ) $`	`Shift`
`$ 0 ( 3 ( 3`	`a ) ) $`	`Shift`
`$ 0 ( 3 ( 3 a 2`	`) ) $`	`Shift`
`$ 0 ( 3 ( 3 L 4`	`) ) $`	`Reduce: L → a`
`$ 0 ( 3 ( 3 L 4 ) 5`	`) $`	`Shift`
`$ 0 ( 3 L 4`	`) $`	`Reduce: L → ( L )`
`$ 0 ( 3 L 4 ) 5`	`$`	`Shift`
`$ 0 L 1`	`$`	`Reduce: L → ( L )`
`$ 0 L′`	`$`	`Reduce: L′ → L`
`$ 0 L′`	`$`	`Accept`

Normally, the parser's actions are controlled by a table. The table for the LR(0) grammar above is the following

State	Action	Rule	Input			GoTo
			(	a	)	L
0	Shift		3	2		1
1	Reduce	L′ → L				
2	Reduce	L → a				
3	Shift		3	2		4
4	Shift				5	
5	Reduce	L → (L)				

The LR(0) Parsing Algorithm

Let S be the current state, which is on the top of the stack.

1. If S contains an item $A \rightarrow \alpha \bullet X \beta$, where X is a terminal, then shift X onto the stack and push the state containing the item $A \rightarrow \alpha X \bullet \beta$.

   If the current input token X does not match any item, $A \rightarrow \alpha \bullet X \beta$, in S, however, declare an error.

2 If S contains any completed item $A \rightarrow \gamma \bullet$, then the action is to reduce by the rule $A \rightarrow \gamma$. That is, pop γ off the stack and push A.

   If the reduction is the rule $S' \rightarrow S$, where S′ is the start state, then—if the input is empty—the action is to accept. Otherwise, declare an error.

   In the other cases, to compute the new state, we backtrack in the DFA until the beginning of the construction of γ, that is to the state containing $A \rightarrow \bullet \gamma$ and for the new state, take the transition for A.

A grammar is said to be LR(0) if the rules above are unambiguous. That is, in Rule 1, there are not two items in a state $A \rightarrow \alpha \bullet X \beta$ and $A \rightarrow \alpha \bullet$. This is called a **shift-reduce** conflict. In Rule 2, a state which contains to completed items $A \rightarrow \alpha \bullet$ and $A \rightarrow \beta \bullet$ is said to have a **reduce-reduce** conflict.

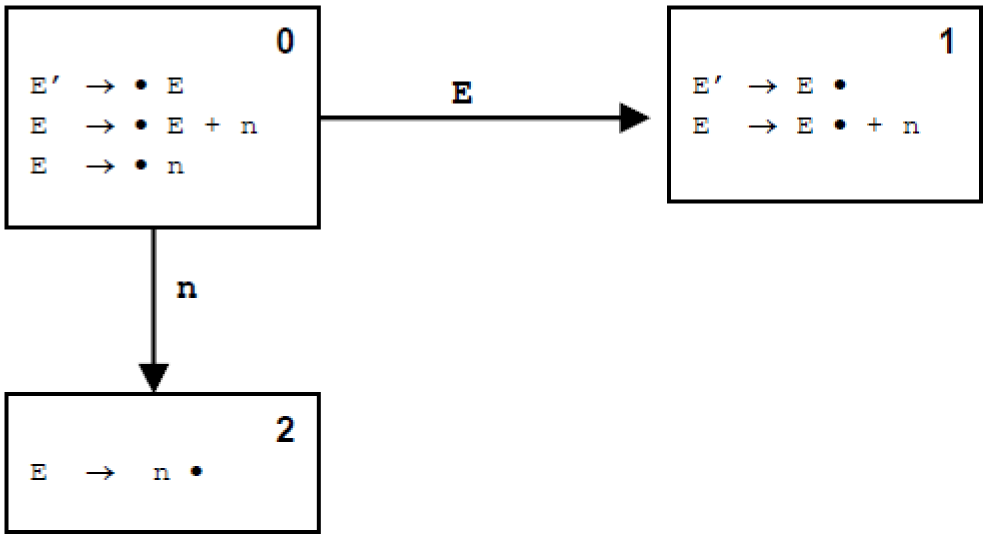
Example of a grammar that is not LR(0)

```
E → E + n | n
```

First augment the grammar with a new start symbol E′. Then find the LR(0) items.

```
E′ →• E

E′ → E •

E →• E + n

E → E • + n

E → E + • n

E → E + n •

E →• n

E → n •
```

Next make the DFA for this grammar



But notice that state 1 has a **shift-reduce** error

SLR(1) Parsing

SLR(1) parsing is Simple LR parsing. It uses one token as a lookahead. It can significantly increase the power of LR(0) parsing. How does it do this?

1 It consults the input token before a shift is made.

2 It uses the Follow set of a non-terminal to decide if a reduction should be made.

SLR(1) Parsing Algorithm

Let S be the current state at the top of the parsing stack.

1. If S contains any item of the form $A \rightarrow \alpha \bullet X \beta$, where X is a terminal and X is the next token in the input string, then the action is to shift the current input token onto the stack and the new state pushed onto the stack is the one containing the item $A \rightarrow \alpha X \bullet \beta$.

2. If state S contains a complete item $A \rightarrow \gamma \bullet$ and the next input token is in Follow(A), then there is a reduction by the Rule $A \rightarrow \gamma$.

   If the reduction is $S' \rightarrow S$, where S is the start state, then the action is to accept the input. To compute the new state, remove the string γ and its corresponding states from the stack, back up in the DFA to the state from which the construction of γ began, push A onto the stack and push the state containing the new item $B \rightarrow \alpha A \bullet \beta$.

3. If the next input token fits neither of these two cases, it is an error.

Example: Create an SLR(1) table for the grammar

```
Decl′ → Decl

Decl → Type Var-List

Type → int | float

Var-List → ID Var-List′

Var-List′ →, Var-List | ε
```

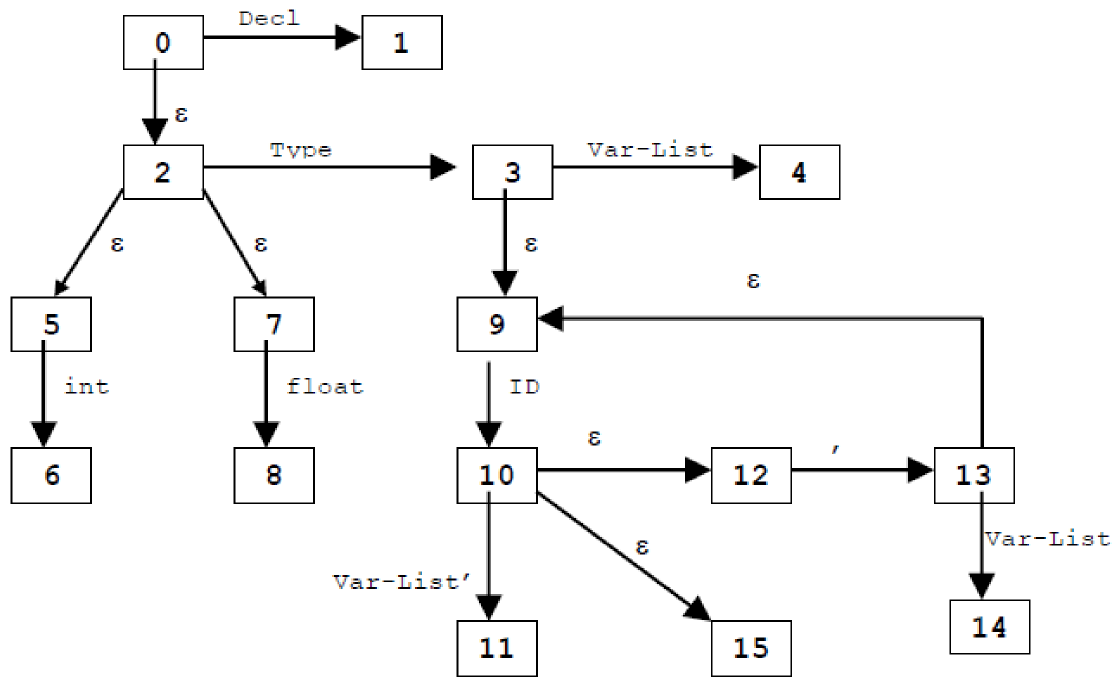
and parse the expression `int ID, ID`.

The first step is to create the items and the Follow sets.

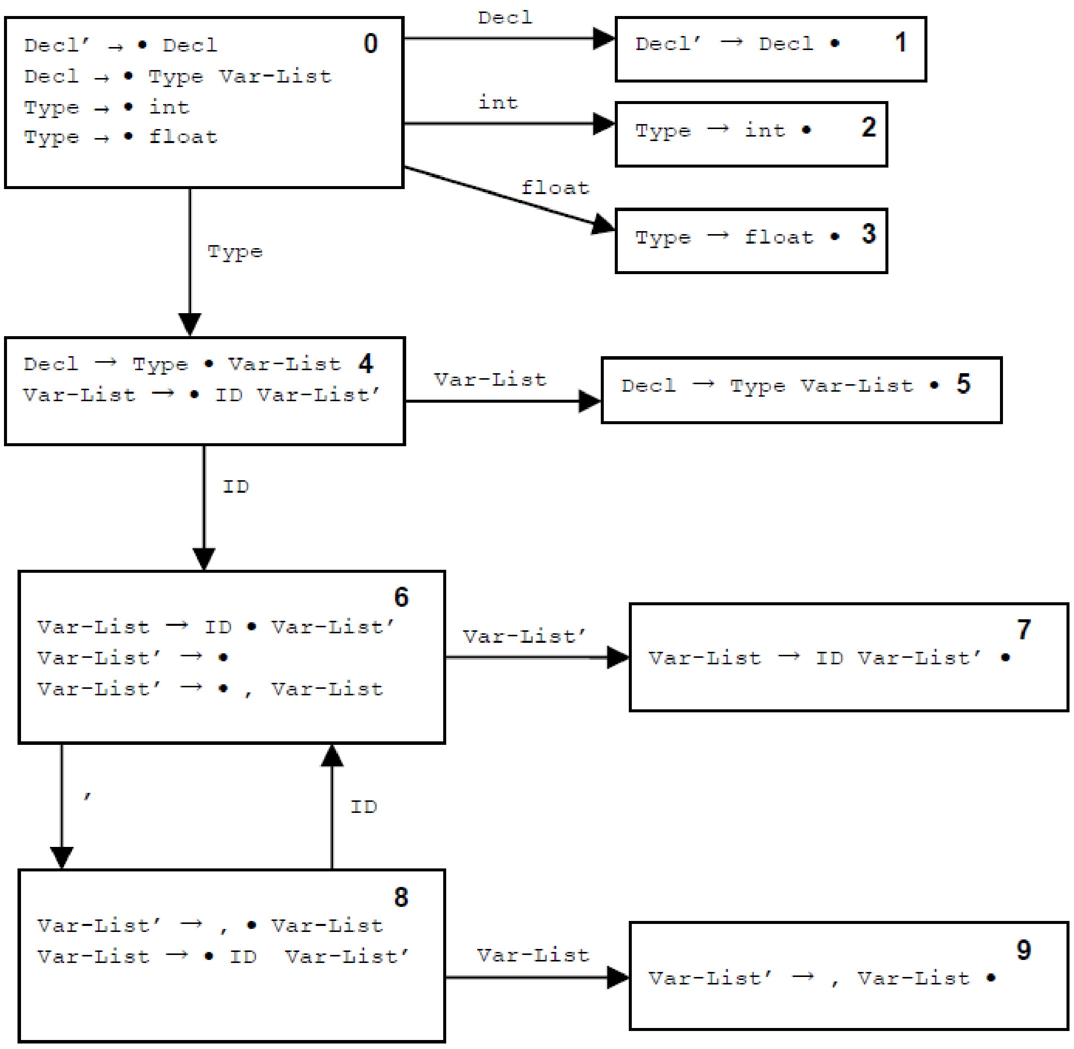
```
 0.  Decl′ → • Decl
 1.  Decl′ → Decl •
 2.  Decl → • Type Var-List
 3.  Decl → Type • Var-List
 4.  Decl → Type Var-List •
 5.  Type → • int
 6.  Type → int •
 7.  Type → • float
 8.  Type → float •
 9.  Var-List → • ID Var-List′
10.  Var-List → ID • Var-List′
11.  Var-List → ID Var-List′ •
12.  Var-List′ → • , Var-List
13.  Var-List′ → , • Var-List
14.  Var-List′ → , Var-List •
15.  Var-List′ → •
```

Follow(`Decl`) = { $ }
Follow(`Type`) = { `ID` }
Follow(`Var-List`) = { $ }
Follow(`Var-List′`) = { $ }
Follow(`Decl′`) = { $ }

The NFA, using only state numbers because of the size, is the following.



The DFA is on the following page.



Decl′ → • Decl
Decl → • Type Var-List
Type → • int
Type → • float
0
Decl
Decl′ → Decl •
1
int
Type → int •
2
float
Type → float •
3
Type
Decl → Type • Var-List
Var-List → • ID Var-List′
4
Var-List
Decl → Type Var-List •
5
ID
Var-List → ID • Var-List′
Var-List′ → •
Var-List′ → • , Var-List
6
Var-List′
Var-List → ID Var-List′ •
7
,
ID
Var-List′ → , • Var-List
Var-List → • ID Var-List′
8
Var-List
Var-List′ → , Var-List •
9

The resulting table is

State	**Input**					**GOTO**			
	int	**float**	**,**	**ID**	**$**	**Decl**	**Type**	**Var-List**	**Var-list′**
0	S2	S3				1	4		
1					R1				
2				R6					
3				R8					
4				S6				5	
5					R4				
6			S8		R15				7
7					R11				
8				S6				9	
9					R14				

The parsing action for the string `int ID, ID` is shown below.

Stack	Input	Action
`$ 0`	`int ID , ID $`	`Initialize`
`$ 0 int 2`	`ID , ID $`	`Shift 2`
`$ 0 Type 4`	`ID , ID $`	`Type → int`
`$ 0 Type 4 ID 6`	`, ID $`	`Shift 6`
`$ 0 Type 4 ID 6 , 8`	`ID $`	`Shift 8`
`$ 0 Type 4 ID 6 , 8 ID 6`	`$`	`Shift 6`
`$ 0 Type 4 ID 6 , 8` $\underline{\texttt{ID 6 Var-List' 7}}$	`$`	`Var-List′ → ε`
`$ 0 Type 4 ID 6` $\underline{\texttt{, 8 Var-List 9}}$	`$`	`Var-List → ID Var-List′`
`$ 0 Type 4` $\underline{\texttt{ID 6 Var-List' 7}}$	`$`	`Var-List′ → , Var-List`
`$ 0` $\underline{\texttt{Type 4 Var-List 5}}$	`$`	`Var-List → ID Var-List′`
`$ 0` $\underline{\texttt{Decl 1}}$	`$`	`Decl → Type Var-List`
`$ 0 Decl′`	`$`	`Decl′ → Decl`
`$ 0 Decl′`	`$`	`Accept`

General LR(1) Parsing

The most general form of LR parsing is called LR(1) parsing or canonical LR(1).

SLR(1) parsing, while using a lookahead token, also uses a DFA that is constructed without using the lookahead token.

LR(1) parsing, however, uses the lookahead as part of the construction of the DFA. This DFA has states that contain **LR(1) items**.

An LR(1) item is a pair that consists of an LR(0) item and a lookahead symbol. For example, $[\ A \rightarrow \alpha \bullet \beta,\ a\ ]$.

Transitions between LR(1) items in an NFA

1. Given an LR(1) item $[\ A \rightarrow \alpha \bullet X\, \beta,\ a\ ]$, where X is any symbol (terminal or non-terminal), there is a transition on X to the item $[\ A \rightarrow \alpha\, X \bullet \beta,\ a\ ]$.

2. Given an LR(1) item $[\ A \rightarrow \alpha \bullet X\, \beta,\ a\ ]$, where X is a non-terminal, there are ε-transitions to items $[\ X \rightarrow \bullet \gamma,\ b\ ]$ for every production $X \rightarrow \gamma$ and for every token b in First($\beta\ a$) .

Note that the item $[\ A \rightarrow \alpha \bullet X\, \beta,\ a\ ]$ says that we are ready to recognize X, but only if X is followed by a string derivable from βa , and these strings must begin with a token from First($\beta\ a$) .

Since β follows X in the production $A \rightarrow \alpha\, X\, \beta$, if a is in Follow(A) then First($\beta\ a$) $\subset$ Follow(X).

The power of an LR(1) parser lies in the fact that First($\beta\ a$) can be a proper subset of Follow(X), whereas an SLR(1) parser uses lookaheads in the entire set of Follow(X).

Note that we must also augment the grammar with a new start symbol. If S is the original start symbol, then $S' \rightarrow S$ is the augmented production and $[S' \rightarrow \bullet\ S,\ \$]$ is the start state for the NFA.

Example: Create the LR(1) parsing table for the grammar.

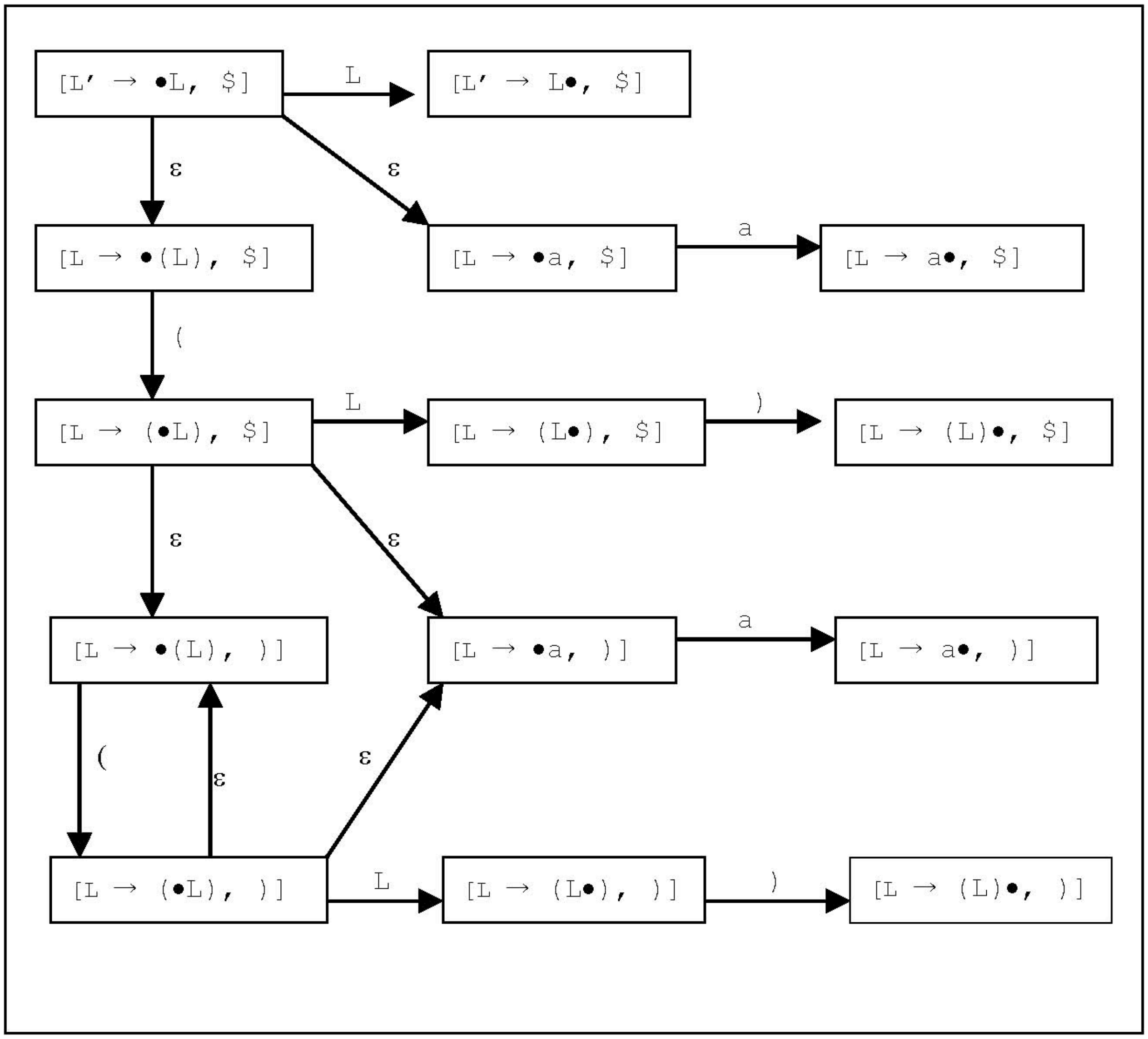
```
L′ → L
L → ( L )
L → a
```

Calculate the LR(0) items, First, and Follow sets.

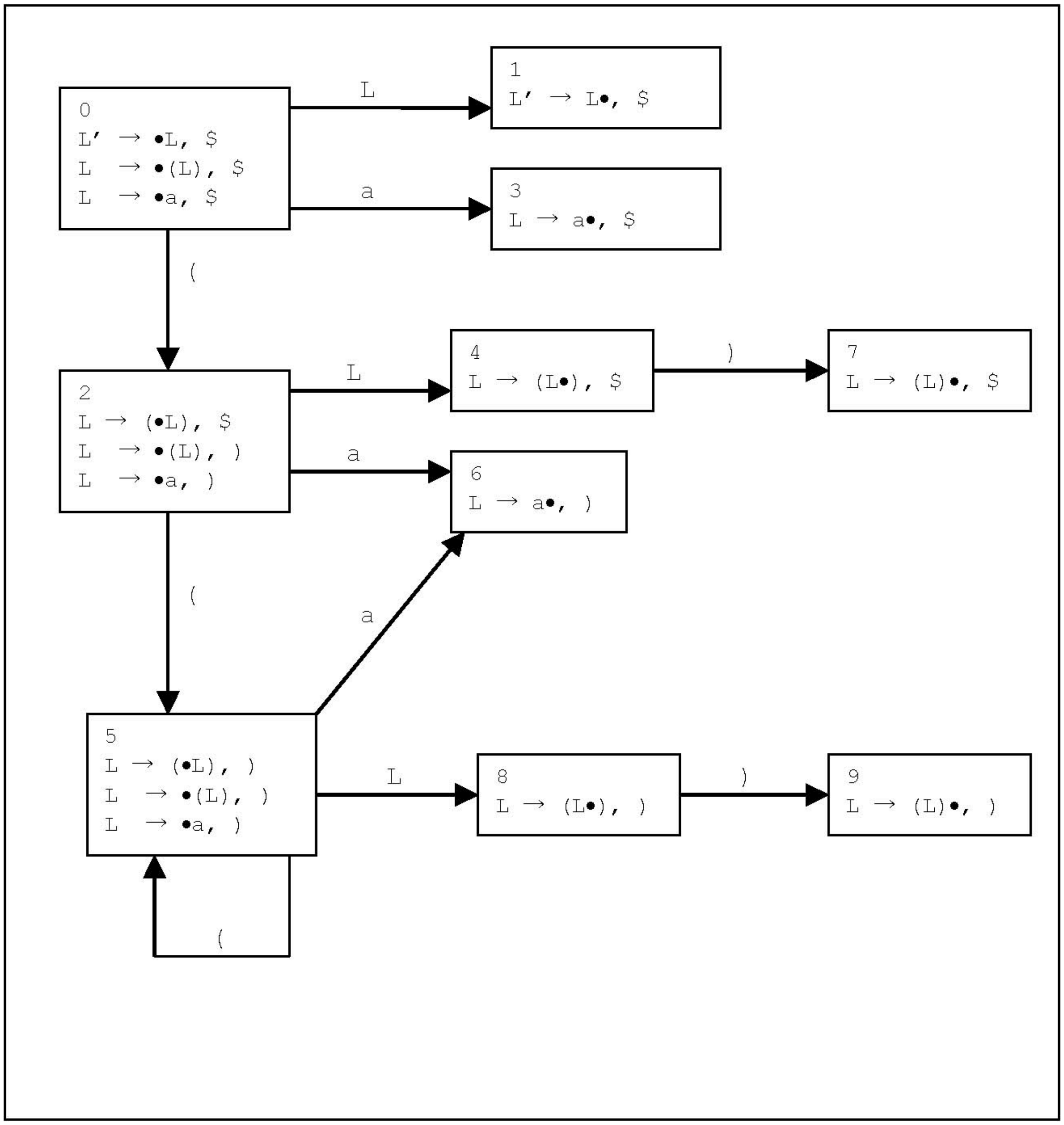
```
L′ →• L
L′ → L •
L →• ( L )
L → (• L )
L → ( L • )
L → ( L ) •
L →• a
L → a •
```

First(`(`) = { `(` }	Follow(`L`) = { `$,`) }
First(`)`) = { `)` }	Follow(`L′`) = { `$` }
First(`a`) = { `a` }	
First(`L`) = { `(, a` }	

The NFA is the following.



The DFA is the following.



Using the DFA above, the parse table for the grammar is

State	INPUT				GOTO
	(	a	)	$	L
0	S2	S3			1
1				Accept	
2	S5	S6			4
3				L → a	
4			S7		
5	S5	S6			8
6			L → a		
7				L → (L)	
8			S9		
9			L → (L)		

The General LR(1) Parsing Algorithm

1. If state S contains any LR(1) item of the form [$A \rightarrow \alpha \bullet X \beta$, a], where X is a terminal and X is the next token in the input string, then the action is to shift X onto the stack and the state that is pushed is the one that contains the item [$A \rightarrow \alpha X \bullet \beta$, a].

2. If state S contains a complete item [$A \rightarrow \alpha \bullet$, a] and a is the next input token, then the action is to reduce by the rule $A \rightarrow \alpha$.

   If the reduction is by the rule $S' \rightarrow S$, then the action is to accept the input string.

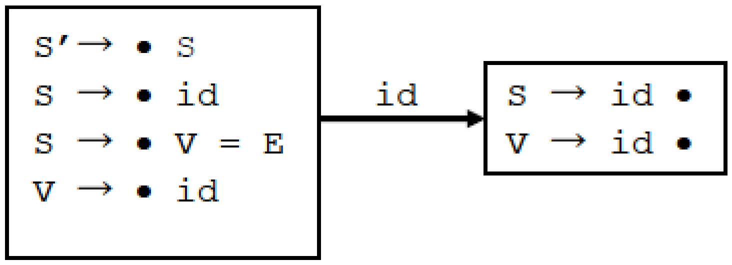
   To compute the new state, pop the string α from the stack. The state uncovered must contain an item of the form [$B \rightarrow \alpha \bullet A \beta$, b]. Push A on the stack and push the state containing the item [$B \rightarrow \alpha\ A \bullet \beta$, b].

3. If the next input token is such that neither of the above cases apply, an error is declared. A grammar is said to be LR(1) if the above rules apply without ambiguity. In particular,

   A. For any item [$A \rightarrow \alpha X \bullet \beta$, a] in state S where X is a terminal, there is no complete item in S of the for [$B \rightarrow \gamma \bullet$, X]. Otherwise, this is a **shift-reduce** conflict.

   b. There are no two items in a state S of the form [$A \rightarrow \alpha \bullet$, a] and [$B \rightarrow \beta \bullet$, a]. Otherwise, this is a **reduce-reduce** conflict.

Example: parse **(a)**



Example: Give

```
S → id | V = E

V → id

E → V | n
```

This grammar was not SLR(1) (see page 216 of the textbook) because there is a transition on `id` to a state containing the items `S` → `id` • and `V` → `id` •. However, this grammar can be parsed by an LR(1) parser.

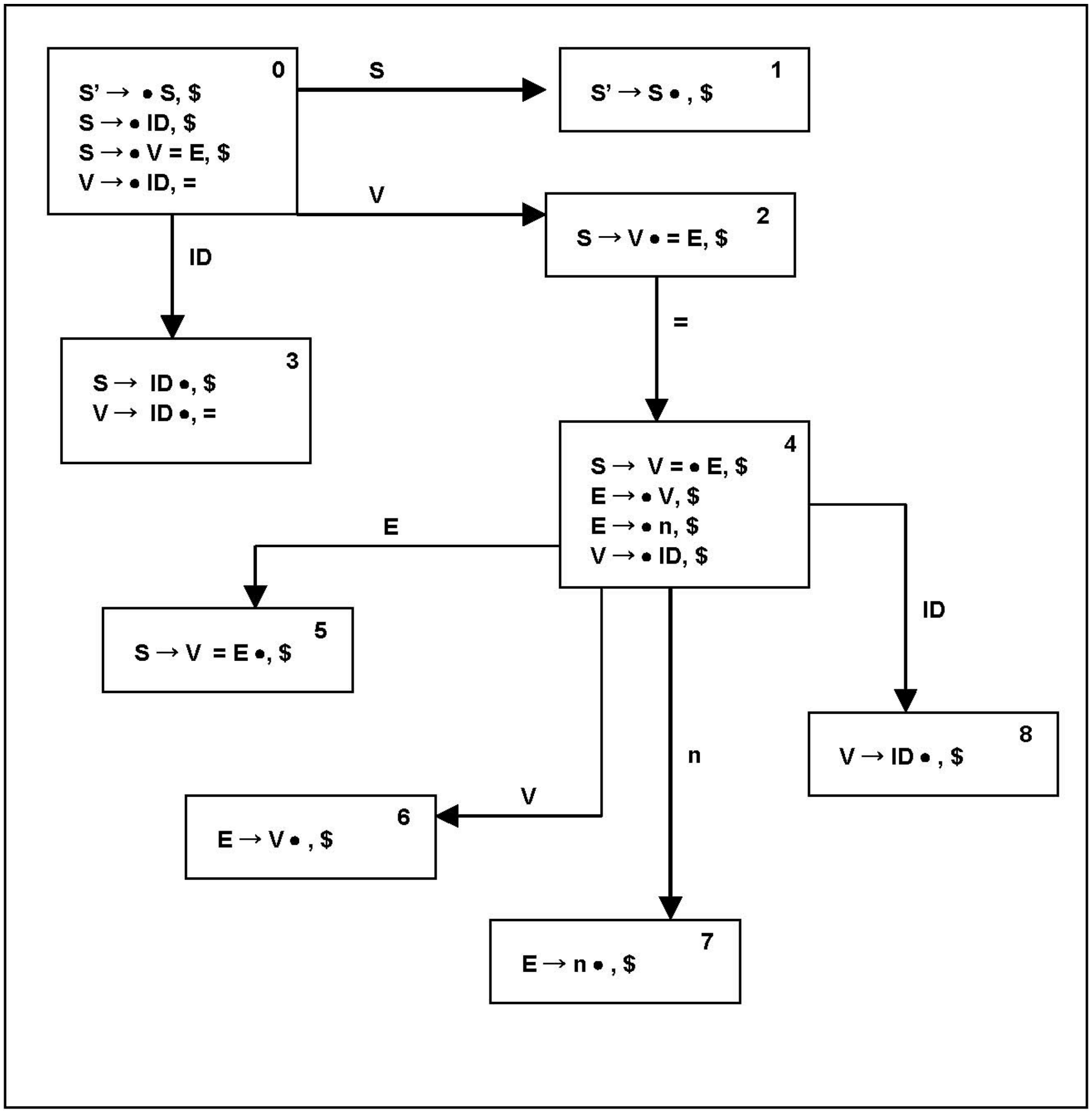
Create the LR(0) items, the NFA, the DFA, and the parse table and parse the expression `id = n`.

After augmenting the grammar with the rule `S′` → `S`, the LR(0) items are:

```
S′→• S
S′→ S•
S →• id
S → id •
S →•V = E
S → V • = E
S → V = • E
S → V = E •
V →• id
V → id •
E →• V
E → V •
E →• n
E → n •

E →• V
```

```
E → V •

First( id ) = { id }          First( S ) = { id }
First( = ) = { = }            First( V ) = { id }
First( n ) = { n }            First( E ) = { n, id }
```

The DFA follows



From the DFA, the parse table is

State	INPUT				GOTO		
	id	=	n	$	S	V	E
0	S3			S4	1	2	
1				Accept			
2		S4					
3		V → E		S → id			
4	S8		S7			6	5
5				S → V = E			
6				E → V			
7				E → n			
8				V → id			

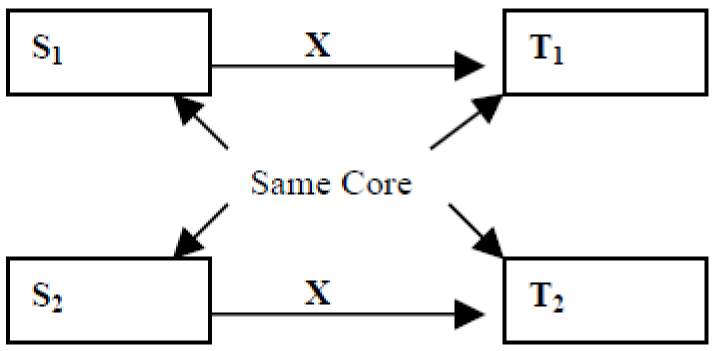
Stack	Input	Action
$ 0	id = n $	Initialize
$ 0 id 3	= n $	S3
$ 0 V 2	= n $	V → id
$ 0 V 2 = 4	n $	S4
$ 0 V 2 = 4 n 7	$ S	7
$ 0 V 2 = 4 E 5	$	E → n
$ 0 S 1	$	S → V = E
$ 0 S 1	$	Accept

LALR(1) Parsing

LALR stands for "Look Ahead LR", and it is based on the observation that the size of an LR(1) DFA can be large because many states can contain the same LR(0) item, but a different lookahead symbol.

Definition: The *core* of a state is the set of LR(0) items in the state.

Given two states S_1 and S_2 of an LR(1) DFA that have the same core, it can be seen that if there is a transition on a symbol X from S_1 to T_1, there will also be a transition from S_2 to a state T_2 on the symbol X, where T_1 and T_2 are states with the same core also.

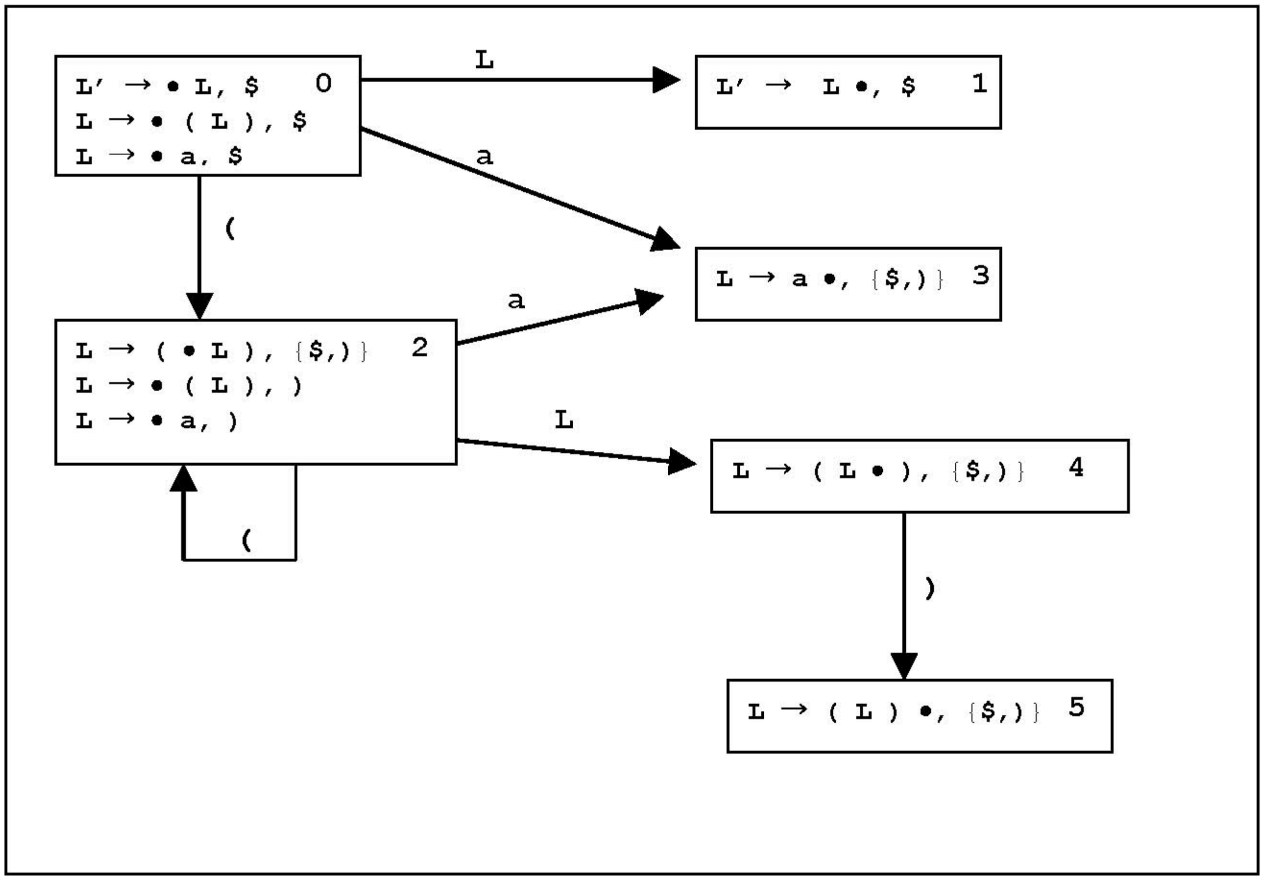


So, LALR(1) parsing allows us to construct a DFA where we collect states that have the same core and union their lookahead sets.

For example, consider the LR(1) DFA created for the grammar

```
L′ → L
L → ( L )
L → a
```

By merging items into states that have the same core, we get a DFA the looks like the following



The parse table can now be constructed as follows

State	**INPUT**				**GOTO**
	(	**a**	**)**	**$**	**L**
0	S2	S3			1
1				Accept	
2	S2	S3			4
3			L → a	L → a	
4			S5		
5			L → (L)	L → (L)	

Now show the parsing action for `( a )`

```
     Stack                                          Input        Action
$ 0                                                ( a ) $      Initialize
$ 0 ( 2                                              a ) $      S2
$ 0 ( 2 a 3                                            ) $      S3
$ 0 ( 2 L 4                                            ) $      L → a
$ 0 ( 2 L 4 ) 5                                          $      S5
$ 0 L 1                                                  $      L → ( L )
$ 0 L 1                                                  $      Accept
```

Chapter 6: *Semantic Analysis*

Semantic analysis is not as cleanly defined as syntactical analysis because of the wide varieties in languages.

One common method to describe the semantics of a program is by the use of an attribute grammar.

An attribute is any property of a programming language construct. Typical examples of attributes are:

* The data type of a variable.
* The value of an expression.
* The location of a variable in memory.
* The object code of a procedure.
* The number of significant digits in a number.

The process of computing an attribute and associating the computed value with a language construct is known as **binding** the attribute. Binding times of an attribute can vary from one language to another. For example, the binding of a type is called **static** binding when the attribute is bound before execution; but it is called **dynamic** binding if it is bound during execution.

In **syntax-directed** semantics, attributes are associated directly with grammar symbols (terminals and non-terminals).

Examples

Grammar Rule	Semantic Rule
digit → 0	digit.val = 0
•••	•••
digit → 9	digit.val = 9
number$_1$ → number$_2$ digit	number$_1$.val → number$_2$.val * digit.val
number → digit	number.val → digit.val

Consider the following two examples: (Overheads for examples 6.2 and 6.3 from the textbook.)

The set of allowable expressions that can appear in an attribute equation is called the **metalanguage** of the attribute grammar. We try to have a metalanguage that is clear enough that we do not have any confusion over its own semantics and close enough to a real programming language that it is easily implemented in a code.

When we try to compute the values for an attribute, we need to pay attention to dependencies. Each grammar rule in an attribute grammar has an associated **dependency graph**. In a dependency graph, an arrow is used to show the direction in which data flows.

Grammar Rule: $\texttt{number}_1 \rightarrow \texttt{number}_2\ \texttt{digit}$

Attribute Equation: $\texttt{number}_1\texttt{.val = number}_2\texttt{.val * 10 + digit.val}$

Dependency Graph:

$$\begin{tikzcd}
& \texttt{number}_1\texttt{.val} & \\
\texttt{number}_2\texttt{.val} \arrow[ur] & \text{Synthesized} & \texttt{digit.val} \arrow[ul]
\end{tikzcd}$$

On the other hand

Grammar Rule: $\texttt{var-list}_1 \rightarrow \texttt{id , var-list}_2$

Attribute Equation: $\texttt{id.dtype = var-list}_1\texttt{.dtype}$
$\texttt{var-list}_2\texttt{.dtype = var-list}_1\texttt{.dtype}$

Dependency Graph

$$\begin{tikzcd}
& \texttt{var-list}_1\texttt{.dtype} \arrow[dl] \arrow[dr] & \\
\texttt{id.dtype} & \text{Inherited} & \texttt{var-list}_2\texttt{.dtype}
\end{tikzcd}$$

Because the dependency graph is based on the attribute equation and the attribute equation is associated with a grammar rule, it is common to superimpose the dependency graph on the parse tree. Then the question becomes, how are these attributes calculated.

We can perform a **topological sort** if the dependency graph has no cycles; however, this is not practical. So, most evaluations use **rule-based** methods. A rule-based method depends on an implicit or explicit traversal of the syntax tree.

Attributes, therefore, are divided into two categories.

An attribute is **synthesized** if the dependencies point from the child to the parent. An example is the first rule above (the `val` example). If all the attributes in an attribute grammar are synthesized, then it is called an S-grammar. A synthesized attribute can be evaluated by a **post-order** traversal of the syntax tree.

```
PostOrder( Treenode N )
   for each child C of N do
        PostOrder( C )

   compute synthesized attributes of N
```

If an attribute is not synthesized, then it is called **inherited**. Dependencies for inherited attributes flow either from parent to child or from sibling to sibling. The second example (the `dtype` example) shows an inherited attribute. Inherited attributes can be evaluated by a **pre-order** traversal of the syntax tree.

```
PreOrder( Treenode N )
  for each child C of N do
     compute all inherited attributes of C

PreOrder( C )
```

However, we must be careful about the order in which children are processed. This is because inherited attributes can flow from sibling to sibling.

Note that if synthesized attributes depend on both inherited and synthesized attributes, but inherited attributes only depend on other inherited attributes, then all attributes can be calculated in one pass of the syntax tree.

```
CombinedOrder( Treenode N )
  For each child C of N do
     compute all inherited attributes of C
     CombinedOrder( C )
     compute all synthesized attributes of C
```

Definition: An attribute grammar for attributes $a_1, \ldots a_k$ is called **L-attributed** if, for each inherited attribute a_j and for each grammar rule $X_0 \rightarrow X_1\ X_2\ \ldots\ X_n$ the value of a_j at X_i depends only on attributes of $X_1 \ldots X_{i-1}$.

Given an L-attributed grammar in which inherited attributes do not depend on the synthesized attributes, a recursive-descent parser can evaluate all the attributes by turning the inherited attributes into parameters to the recursive calls and the synthesized attributes into return values of the recursive calls.

Donald Knuth (1968) showed that all inherited attributes of an attribute grammar can be changed into synthesized attributes by suitable modifications of the grammar without changing the language of the grammar. Doing this, however, may make the grammar more complex. Therefore, it is rarely done. However, if the computation of an attribute seems very difficult, it may be an indication that the grammar in not defined in a suitable way and a change in the grammar may be worthwhile.

The Symbols Table

The symbols table is a major data structure in a compiler. Depending of the organization of the compiler, most of the various phases can make use of the symbols table.

The symbols table can serve as a central database for information on attributes. Depending on the language to be compiled and the language used as the compiler implementation language, the scanner and the parser may make use of the symbols table or it may just be the checker and code generator that makes use of the symbols table.

As a data structure, the symbols table is a dictionary structure. We need to implement the operations `insert`, `delete`, and `lookup` efficiently.

Often in a production compiler, the symbols table is implemented as a hash table. Note that care must be taken to determine the size of the table and the function used for the hash function. In particular, the hash function must make use of an entire identifier to compute an address (how often do we use identifiers such as `temp1`, `temp2`, etc.?) and to account for identifiers such as `tempx` and `xtemp`.

Information that might be stored in a symbols table includes

<u>Information about declarations</u>.

There are four basic types of declarations

1. Constants
2. Type declarations
3. Variable declarations
4. Function, procedure, or method declarations

It is often easier to have a single table to hold all of these, but it may sometimes be necessary to make multiple tables, particularly if the language allows names to be reused. That is, a method and a variable within the method may both have the same name.

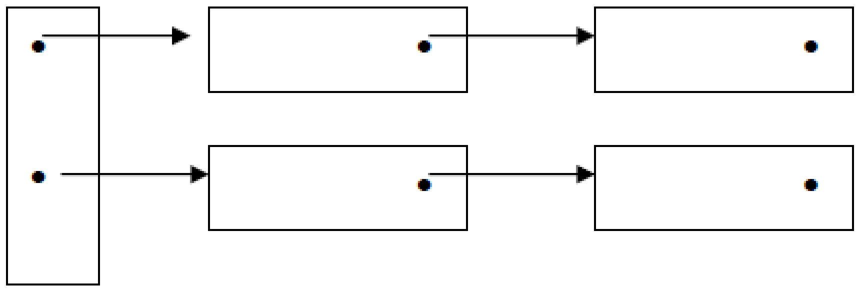
<u>Information about scope and block structure</u>.

Many languages require that a name be declared before it is used. This rule allows a symbols table to be built as parsing proceeds. This means that any time a name is encountered, it can be checked in the symbols table. This is the reason languages such as C have prototypes and Pascal has forward declarations.

Block structure

A **block** is a language construct that allows declarations. For example a method or a pair of curly braces in Java form a block. This gives us scoping rules.

How should a symbols table implement a block structure? First, when a new scope is created, previous declarations must not be overwritten, only hidden. One implementation method of this is a hash table with buckets.



Using this, new declarations are added to the head of the list. The problem with this approach, however, is what to do about collisions.

Another option is to use separate tables for each block level, with a link from table to table.

For our project, an array implementation will be sufficient for the symbols table. In this project, the parser will build a syntax tree, and the checker will take the tree and build the symbols table. The checker will maintain a global variable called `blockLevel`, which is initialized to 0. When a block is entered, `blockLevel` is incremented, and when a block is exited, `blockLevel` is decremented.

Other Comments On The Checker

As the tree is traversed, we need to keep track of declaring or not declaring for variables and function identifiers.

When in declaring mode and an identifier is inserted into the table, we must walk backwards in the table until the block level changes. If the identifier is encountered in this walk, an error is declared: The identifier has already been declared. When an identifier is inserted into the table, either an array or a variable identifier, an unique value must be chosen for the rename field. When a function is declared, its name is entered into the symbols table also, but it does not need a rename value: The function's name will become a label in the assemble language file. However, we do need to record the function's return type in the symbols table.

The checker for this project only needs to check that a `void` function does not contain a `return` statement and that an `int` function does contain a `return` statement. I created a separate visitor object for this purpose.

Type checking for this project only requires verification that a `void` function is not used in an expression.

Building the symbols table is essentially a pre-order (top-down) traversal of the syntax tree. Type checking is essentially a post order (bottom up) traversal.

Given an expression node, recursively evaluate the left and right children and compare the values.

When creating a symbol table, we need to keep track of **declaring mode** or **checking mode**.

In declaring mode, values are inserted into the table. In checking mode, we make sure that something has been declared. When inserting into the table, give each variable a unique rename value.

Checker

1. Variables, arrays and functions are declared before they are used.

2. Variables, arrays and functions cannot be declared twice in the same block.

3. Variables, arrays and functions cannot have the same name within the same block.

4. A `VOID` function does not have a `return` statement.

5. An `INT` function does have a `return` statement.

6. A `VOID` function cannot be used in an expression.

7. An array, when used within the same block as its declaration, is within bounds.

8. An array, when used within an expression, must have a location, and a variable cannot be used as an array.

Chapter 7: *Runtime Environments*

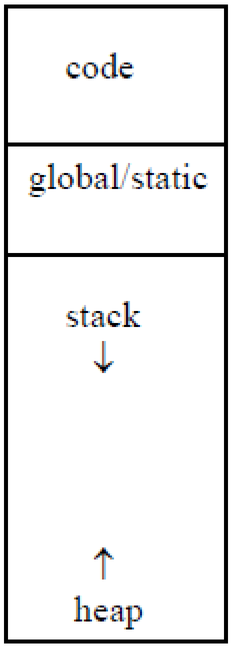
Part of the compilers job is to emit the code necessary to maintain the runtime environment.

Typically, an executable file is divided into two parts: the data area and the code area.

The code section contains the instructions for all procedures/functions. Note that the entry point for these functions can all be calculated at compile time. The entry points are normally expressed as offsets from the start of the code area.

The data area contains global or static data such as constants. As with functions, the addresses of these are all known (or computed) at compile time.

When a program is loaded into memory, storage is allocated for two more areas, the stack and the heap. The stack is for static storage and the heap is for dynamic storage.



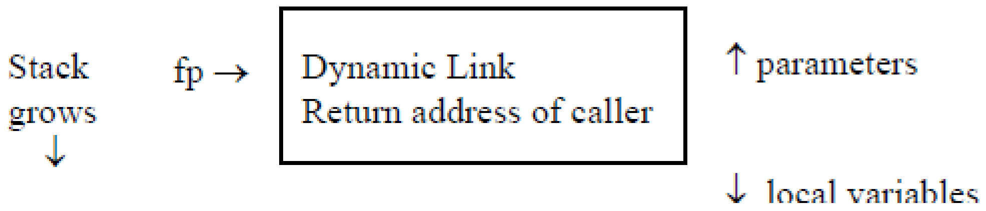
Activation Records or Stack Frames

An important data structure used for static memory is the **activation record**, which is sometimes called a stack frame. Whenever a subroutine is called and activation record is created for the call. An activation record will contain space for:

- Arguments that are passed to it.
- Local variables.
- Compiler generated local temporary variables.
- Bookkeeping data such as static and dynamic links, and the return address of the caller.
- Saved register values

Since an activation record is associated with a subroutine, if function A calls function B, then part of the code that the compiler emits for the call must copy the values of any arguments that are being passed from the activation record of A to the activation record of B. It must also allocate space in the activation record for B for local variables, and it must create a pointer from the activation record of B to the activation record of A. This pointer is sometimes called the **dynamic link** or the **control link**. The return address of function A must be placed in the activation record also.

Often, the activation record for the currently active function is located by a pointer— often called the **frame pointer** or **fp**. This pointer may often point into the middle of the activation record.



The Calling Sequence

1. Compute the arguments and push them onto the runtime stack.
2. Push the `fp` as the dynamic link of the new activation record.
3. Change the `fp` to point to the new activation record; i.e. copy the **stack pointer** or `sp` to `fp`.
4. Store the return address of the caller in the new activation record.
5. Jump to the code of the function to be called.

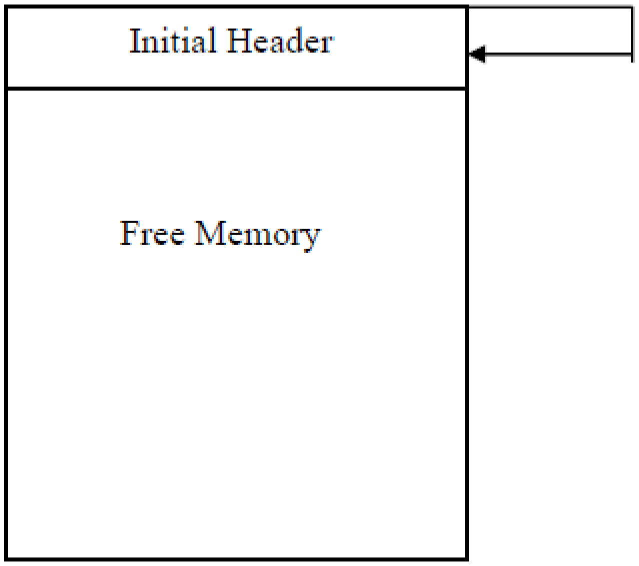
The Exit Sequence

1. Copy the return value—if there is one—back to the caller's activation record.
2. Copy the `fp` to the `sp`
3. Copy the dynamic link to the `fp`.
4. Perform a jump to the return address.
5. Move the `sp` to pop the arguments of the stack.

Local Procedures

Some languages allow a declaration of a procedure that is local to another procedure. The "local" procedure would normally have access to the variables of its parent. This is accomplished with a **static pointer** or sometimes called an **access link**. This is another pointer like a dynamic link, which is placed in the activation record of a function, but it points to the activation record of the lexically enclosing function.

When declarations are nested, and a function is referencing a distance ancestor, several



point sing the difference in block levels
to set **cess chaining**. Or it can be done
using

Dynamic

Dyna here are, of course, many ways to
imple l. In this implementation, the
heap a header at the beginning of the
block ter called the **heap pointer** or **`hp`**.

Consider a header data structure with three fields.

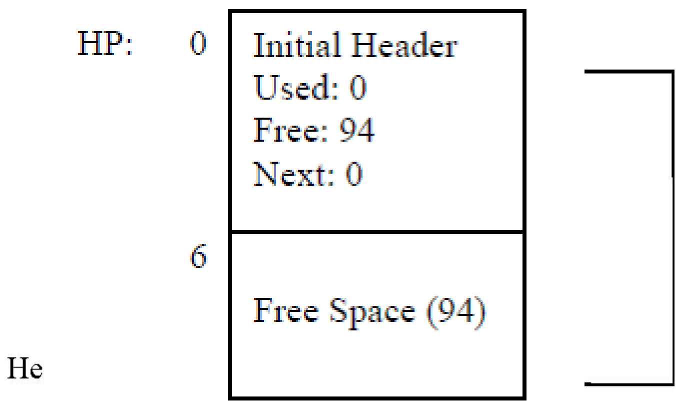
- **`Next`**: This is a pointer to the next header in the list.
- **`Used Size`**: This field holds the size of the used memory in this block.
- **`Free Size`**: This field holds the size of any free memory in this block.

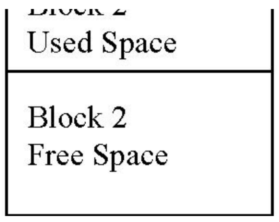
The initial configuration is below. The initial header points to itself as the next block. The *used size* is 0 and the *free size* is the size of the heap minus the size of the initial header.

Heap Pointer →

When there is a call to **`allocate( size )`**, if the free space in the block pointed to by **`hp`** is greater than **`size`** + header size, then:

A new header is created in the beginning of the free space. The **`next`** pointer of the new header points back to initial header, and the **`next`** pointer of the current header is changed to point to the new header. The **`used size`** of the new header contains **`size`** and the **`free size`** of the new header contains the **`free size`** of the previous header minus the header size and minus the allocated **`size`**. The heap pointer (**`hp`**) is then set to point to the new header.





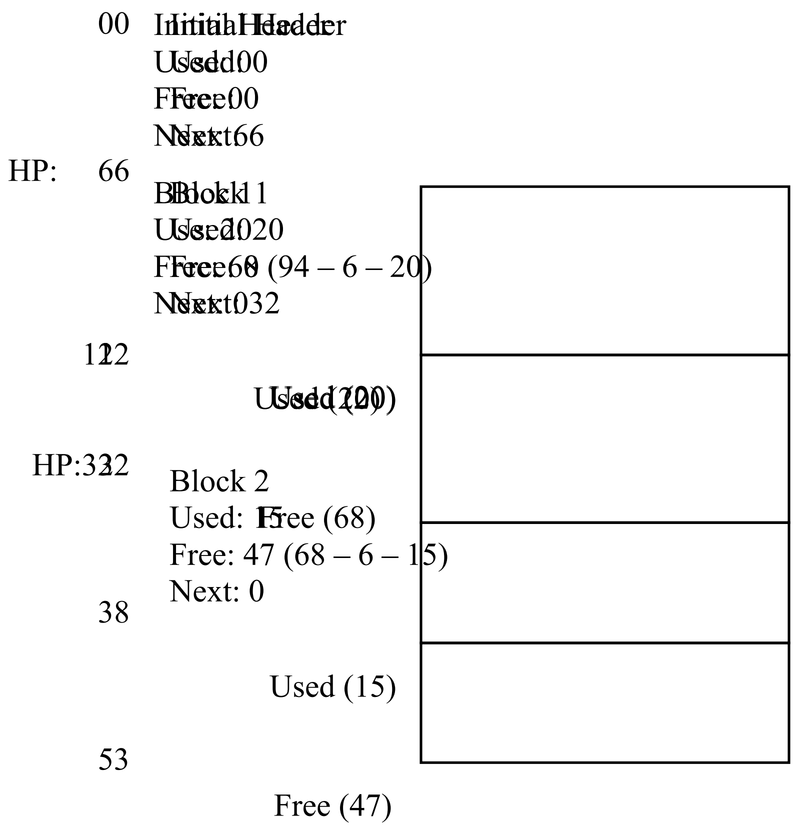
When there is a call to **`free( block pointer )`**, starting from the initial header, the list is traversed until **`block pointer`** is found. The traversal stops at the header before **`block pointer`**. The header size, **`used size`** and **`free size`** of the freed block are added to the **`free size`** of the previous block and the block is removed from the list by assigning the **`next`** field of the previous block the value in the **`next`** field of the freed block.

Consider the following example. A header is 6 bytes and there is 100 bytes altogether for the heap. The call sequence is

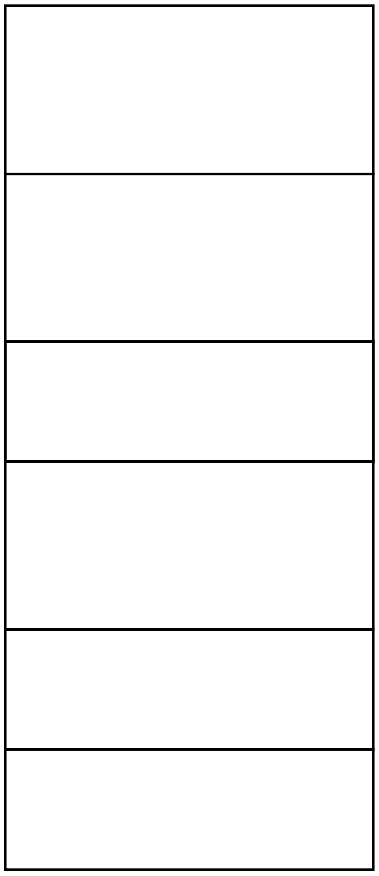
```
1. allocate( 20 )
2. allocate( 15 )
3. allocate( 30 )
4. free( block 1 )
```

The initial configuration is

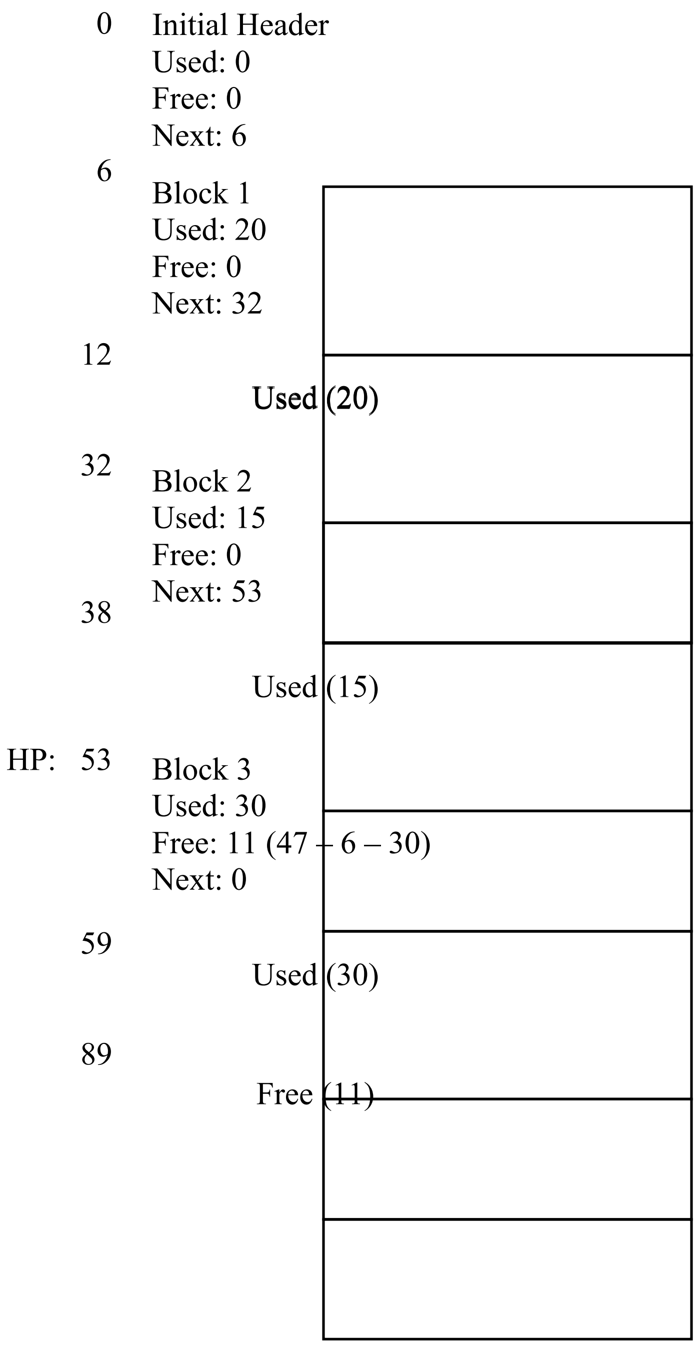
`Allocate( 20 )`

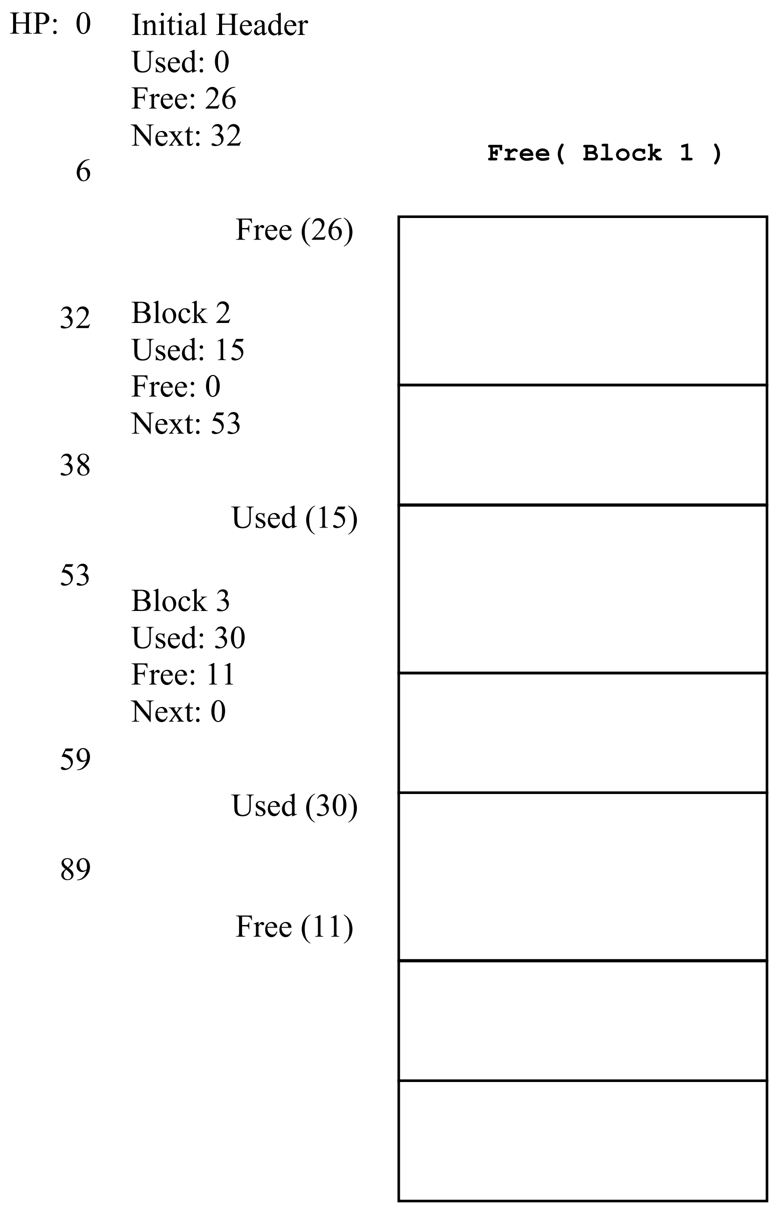


Allocate(15)



Allocate(30)


0
Initial Header
Used: 0
Free: 0
Next: 6
6
Block 1
Used: 20
Free: 0
Next: 32
12
Used (20)
32
Block 2
Used: 15
Free: 0
Next: 53
38
Used (15)
HP: 53
Block 3
Used: 30
Free: 11 (47 – 6 – 30)
Next: 0
59
Used (30)
89
Free (11)



Note that there are three headers, which accounts for 18 bytes; there are two used area, 15 + 30 = 45 bytes; and, there are two free areas, 26 + 11 = 37 bytes. Adding these together gives us 18 + 45 + 37 = 100.

Garbage Collection

The `free` method can leave the heap memory quite fragmented. There are a couple of techniques to perform garbage collection.

Mark and Sweep

First all pointers into the heap are followed, and any heap block that is reached is marked. This requires extra space in the header. Note that this is essentially a backtracking algorithm because there might be pointers in the heap block that lead to other heap blocks. All of these pointers must be followed. Once all pointers have been followed and all reachable blocks have been marked. The sweep phase begins.

The sweep phase begins from the initial header and traverses the heap in a linear order. Any unmarked block is freed, perhaps by linking the block into a free block list.

Mark and sweep is often accompanied memory compactification to move all the free blocks to the end of the heap space. That is to coalesce all free blocks into one free block.

Mark and sweep requires more space in the headers and more time for the 2 passes: the mark pass and the sweep pass. And, the compactification can take much time as well.

Stop-And-Copy

This is sometimes called **Two-Space** garbage collection. In this technique the heap space is divided into two halves and only one half is used at a time. During the sweep pass, a reachable block is automatically moved to the other half. With will also automatically compact the free space at the end of the half. The run time is linear; however, the major disadvantage is that the heap space is cut in half.

Generational Garbage Collectors

Recently, generational garbage collectors have been used. It has been noted, particularly in object-oriented languages, that dynamic objects usually go out of scope very quickly or they survive for a long time. In a generational garbage collection system, new objects are allocated from a small area known as the nursery. If an object survives for a period of time, the object is copied to a more permanent area of memory. The advantage is that a smaller area—just the nursery—needs to be regularly scanned for objects that are no longer reachable.

A generational system can also be set up so that there are three areas of memory: one for the nursery, one for middle aged objects, and one for old age objects. The middle area is scanned less frequently than the nursery, and an object that survives the nursery and the middle age is moved to the old age area where it remains for the duration of the program.

Some Parameter Passing Techniques

Pass By Value

This is the standard in many languages. A copy of the contents of the argument from the caller is made and placed in the activation record of the callee. Any changes the callee makes to the value are not reflected in the caller.

Pass By Reference

Here the arguments must be variables with allocated space. It is the address of the argument that is placed into the corresponding parameter. In effect, the reference parameter becomes an alias for the caller's argument.

Pass By Value-Result

This is similar to pass by reference except no alias is actually established.

The value of the argument is copied to callee. The argument is used, and, at the end of the function call, the value is copied back to location of the caller's argument. In many situations, *pass by value-result* has the same effect as *pass by reference*. However, consider the following example.

```
void p( int x, int y {
   x++;

   y++;
}
```

Then,

```
int a = 1;
p( a, a );
```

When *pass by reference* is used, **a** has the value 3 after the call to **p**. But, if *pass by value-result* is used, **a** has the value 2 after the call to **p**.

Pass by value-result requires several changes to the basic calling sequence.

1. The activation record cannot be freed by the callee, since values in the callee's activation record must be copied back to the caller.

2. The caller must record the addresses of the arguments as temporary variables or must recompute the address on the return from the called procedure.

Pass By Name (Delayed Evaluation)

The idea is that a parameter is not evaluated in the callee until its actual use. Thus a textual representation at the point of call replaces the name of the parameter. Consider the following.

```
int i;
int a[ 10 ];


void p( int x ) {
   i++;
   x++;

}

i = 1;
a[ 1 ] = 1;
a[ 2 ] = 2;


p( a[ i ] );
```

What is the result? `a[ i ]` replaces `x` in the function `p`, so at the time of the statement `x++`, the instruction is `a[ 2 ]++`. Therefore `a[ 2 ]` is 3 and `a[ 1 ]` remains at 1.

This was first offered as a parameter passing mechanism in Algol60. But, it was not really popular because it can give counter-intuitive results. Also, it can be difficult to implement because each argument must be turned into a type of procedure call often referred to as a **thunk**. And, the use of one argument may result in several evaluations, where the argument has a different value each time.

A variation of this is the language construct used in some functional languages such as *Haskell* and *SaSL* called **lazy evaluation**. Using this, for example, *SaSL* can represent a sequence as a variable.

Survey of Code Optimization Techniques (Section 8.9)

Areas For Improvement

Improvements can involve both making code run faster and using less space.

Register Allocation

We want to keep data that we use frequently in registers, but there are only so many registers.

Since the IBM 801 project in 1975, register allocation has been done using a graph

```
      contents
      popl 0, -2
      pushl 0, -1
      popl 0, -3
      branch L3
L2:
L3:   pushl [illegible], -1
      pushc 1
```

coloring algorithm. If register allocation is not done well, there is a penalty of *spills* and *fills*. Register allocation is an area where even a slight improvement can have a large effect on performance.

Unnecessary Operations

A large category for improvement is to eliminate operations that are duplicated or are unnecessary. For example, common subexpression elimination is when an expression that always has the same result appears many times in a program. The result can be calculated once and saved in a temporary variable. Then, the expression can be replaced with the result.

Another aspect of this is the elimination of unreachable code. One variation is *jump optimization*. See the **`SelectionSort.asm`** example.

In the example above, there is a branch (an unconditional statement) to label L3. However, L3 represents the next address, so the branch statement could be removed from the code without any adverse effect.

Costly Operations

Reduction in Strength is a technique of replacing an expensive—in terms of CPU cycles—operation with a less costly operation.

For example, rather than a multiply by 2, we could use a left shift operation. Or, rather than a raising to a small power such as **`pow( x, 3 )`**, a statement such as **`x * x * x`** could be used. Or, instead of **`5 * x`**, **`4 * x + x`** could be used where **`4 * x`** is implemented by a left shift by two bits.

Constant Folding is a technique of replacing constant expressions with a constant that is the value of the expression. That is, the expression can be evaluated by the compiler; there is no need to output code to evaluate the expression.

Code Inlining is a technique of taking short functions and replacing the call to the function with the body of the function. An example might be a function which performs the swap of its two arguments. Code inlining increases the size of the code, but can reduce the runtime, particularly if the function is called frequently.

Eliminating Tail Recursion Consider the following example that uses the greatest common divisor algorithm of Euclid.

```
int gcd( int u, int v ) {
   if( v == 0 ) {

        return u;
   }
    else {
        return gcd( v, u % v );
    }
}
```

becomes

```
int gcd( int u, int v ) {
   begin:
     if( v == 0 ) {
        return u;
     }
     else {
        int t1 = v;
        int t2 = u % v;
        u = t1;
        v = t2;
        goto begin;

     }
}
```

Loop Unrolling involves a reduction in the number of iterations that a loop makes. Since a comparison is an expensive operation, reducing the number of iterations reduces the number of comparisons. For example, given the following:

```
for( int i = 0; i < 100; i++ ) {
   a[ i ] = i;
   }
```

becomes

```
for( int i = 0; i < 10; i++ ) {
   int b = i * 10;
   a[ b ] = b;
   a[ b + 1 ] = b + 1;
   a[ b + 2 ] = b + 2;
   a[ b + 3 ] = b + 3;

         •
         •
         •

   a[ b + 9 ] = b + 9;
 }
```

Many times optimizations need to be performed iteratively as one optimization may uncover other possibilities. For example:

```
x = 1;
   •••
y = 0;
   •••
if( y ) x = 0;
   •••
if( x ) y = 1;
```

In the first iteration, because `y = 0` we get `if( 0 ) x = 0`. However, this makes `x = 0` unreachable, thus the code is eliminated, and we get,

```
x = 1;
   •••
y = 0;
   •••
if( 1 ) y = 1;
```

But now it becomes.

```
x = 1;
   •••
y = 0;
   •••
y = 1;
```

Classifying Optimizations By Areas

The categories over which optimizations may be carried out are

- **Local**. These are optimizations that apply to straight line segments of code where there are no jumps into or out of the segment. These are also called **basic blocks**.

- **Global**. These are optimizations that are limited to subroutines—despite the name global.

- **Interprocedural**. These are optimizations that extend beyond subroutine boundaries.

Local optimizations are simpler to make than the others because there are no side effects. A variable given a value at the beginning of a block has that value—in the absence of a reassignment—at the end of the block. But a jump into the middle of the block may change the value.

Global optimizations are more difficult to implement and generally require the use of a technique called Data Flow Analysis, which attempts to collect data across jump boundaries.

Interprocedural analysis can rarely be performed, especially in the case of a program spread across multiple files.

The basic data structure used for data flow analysis is called a **data flow graph**. This is a graph whose nodes are **basic blocks**.

A basic block is a set of instructions that are executed linearly. That is, there is no jump into or jump out of the block. A block is defined as

1 The first instruction begins a new block.
2 Each label that is the target of a jump begins a new block.
3 Each instruction that follows a jump begins a new block.

In some cases a basic block consists of only 1 instruction. In other cases, it is possible that subroutine calls are embedded in a basic block.

The following section from SelectionSort.asm illustrates basic blocks.

```
minloc:       pushc 0       ;local variable tempVar000004
              pushc 0       ;local variable tempVar000005
              pushc 0       ;local variable tempVar000006
              pushl 0, 2
              popl 0, -3
              pushl 0, 3
              pushl 0, 2
              sub
              contents
              popl 0, -2
              push1 0, 2
              pushc 1
              add
              popl 0, -1
```

```
L0:           pushl 0, 1
              Pushl 0, -1
              less
              breql L1
```

```
              pushl 0, -2
              pushl 0, 3
              push 0, -1
              sub
              contents
              less
              breql L2
```

```
              pushl 0,3
              pushl 0,-1
              sub
              contents
              popl 0,-2
              pushl 0, -1
              popl 0,-3
              branch L3
```

```
L2:
L3:           pushl 0, -1
              Pushc 1
              add
              popl 0, 1
              branch L0
```

```
L1:
              pushl 0, -3
              freturn 3
```

In a flow graph, the basic blocks form the nodes and the edges of the graph represent jumps, either conditional or unconditional.

One standard use of a data flow graph is to determine the **reaching definitions** of a variable. A *definition* is an instruction that sets the value of a variable, such as an assignment statement or a read statement. A definition is said *to reach a block* if, at the beginning of the block the variable may have the same definition; i.e. there is a path through the graph from the basic block which holds the definition to the block which is reached.

Reaching definitions can be used in a number of different optimization techniques. For example, they can be used in constant propagation. If the only definition of a variable reaching a block is a single constant value, then the variable can be replaced by this value, at least until another definition for the variable occurs in the block.

Reaching definitions can also be used to help determine which variable values should be kept in registers. As part of data flow analysis, we can do **liveness analysis**. Within a reaching definition, a variable is live if it contains a value that may be needed later.