Top-Down Parsing

Handout written by Maggie Johnson and revised by Julie Zelenski.

Possible Approaches

The syntax analysis phase of a compiler verifies that the sequence of tokens extracted by the scanner represents a valid sentence in the grammar of the programming language. There are two major parsing approaches: *top-down* and *bottom-up*. In top-down parsing, you start with the start symbol and apply the productions until you arrive at the desired string. In bottom-up parsing, you start with the string and reduce it to the start symbol by applying the productions backwards. As an example, let's trace through the two approaches on this simple grammar that recognizes strings consisting of any number of a's followed by at least one (and possibly more) b's:

```
S  ->  AB
A  ->  aA | ε
B  ->  b | bB
```

Here is a top-down parse of aaab. We begin with the start symbol and at each step, expand one of the remaining nonterminals by replacing it with the right side of one of its productions. We repeat until only terminals remain. The top-down parse produces a leftmost derivation of the sentence.

```
S
AB          S -> AB
aAB         A -> aA
aaAB        A -> aA
aaaAB       A -> aA
aaaεB       A -> ε
aaab        B -> b
```

A bottom-up parse works in reverse. We begin with the sentence of terminals and each step applies a production in reverse, replacing a substring that matches the right side with the nonterminal on the left. We continue until we have substituted our way back to the start symbol. If you read from the bottom to top, the bottom-up parse prints out a rightmost derivation of the sentence.

```
aaab
aaaεb       (insert ε)
aaaAb       A -> ε
aaAb        A -> aA
aAb         A -> aA
Ab          A -> aA
AB          B -> b
S           S -> AB
```

In creating a parser for a compiler, we normally have to place some restrictions on how we process the input. In the above example, it was easy for us to see which productions were appropriate because we could see the entire string aaab. In a compiler's parser, however, we don't have long-distance vision. We are usually limited to just one-symbol of *lookahead*. The lookahead symbol is the next symbol coming up in the input. This restriction certainly makes the parsing more challenging. Using the same grammar from above, if the parser sees only a single b in the input and it cannot lookahead any further than the symbol we are on, it can't know whether to use the production B –> b or B –> bB.

Backtracking

One solution to parsing would be to implement *backtracking*. Based on the information the parser currently has about the input, a decision is made to go with one particular production. If this choice leads to a dead end, the parser would have to backtrack to that decision point, moving backwards through the input, and start again making a different choice and so on until it either found the production that was the appropriate one or ran out of choices. For example, consider this simple grammar:

S –> bab | bA
A –> d | cA

Let's follow parsing the input bcd. In the trace below, the column on the left will be the expansion thus far, the middle is the remaining input, and the right is the action attempted at each step:

S	bcd	Try S –> bab
bab	bcd	match b
ab	cd	dead-end, backtrack
S	bcd	Try S –> bA
bA	bcd	match b
A	cd	Try A –> d
d	cd	dead-end, backtrack
A	cd	Try A –> cA
cA	cd	match c
A	d	Try A –> d
d	d	match d
		Success!

As you can see, each time we hit a dead-end, we backup to the last decision point, unmake that decision and try another alternative. If all alternatives have been exhausted, we back up to the preceding decision point and so on. This continues until we either find a working parse or have exhaustively tried all combinations without success.

A number of authors have described backtracking parsers; the appeal is that they can be used for a variety of grammars without requiring them to fit any specific form. For a

small grammar such as above, a backtracking approach may be tractable, but most programming language grammars have dozens of nonterminals each with several options and the resulting combinatorial explosion makes a this approach very slow and impractical. We will instead look at ways to parse via efficient methods that have restrictions about the form of the grammar, but usually those requirements are not so onerous that we cannot rearrange a programming language grammar to meet them.

Top-Down Predictive Parsing

First, we will focus in on top-down parsing. We will look at two different ways to implement a non-backtracking top-down parser called a *predictive parser*. A predictive parser is characterized by its ability to choose the production to apply solely on the basis of the next input symbol and the current nonterminal being processed. To enable this, the grammar must take a particular form. We call such a grammar *LL(1).* The first "L" means we scan the input from left to right; the second "L" means we create a leftmost derivation; and the 1 means one input symbol of lookahead. Informally, an LL(1) has no left-recursive productions and has been left-factored. Note that these are necessary conditions for LL(1) but not sufficient, i.e., there exist grammars with no left-recursion or common prefixes that are not LL(1). Note also that there exist many grammars that cannot be modified to become LL(1). In such cases, another parsing technique must be employed, or special rules must be embedded into the predictive parser.

Recursive Descent

The first technique for implementing a predictive parser is called *recursive-descent*. A recursive-descent parser consists of several small functions, one for each nonterminal in the grammar. As we parse a sentence, we call the functions that correspond to the left side nonterminal of the productions we are applying. If these productions are recursive, we end up calling the functions recursively.

Let's start by examining some productions from a grammar for a simple Pascal-like programming language. In this programming language, all functions are preceded by the reserved word FUNC:

```
program        ->    function_list
function_list  ->    function_list function | function
function       ->    FUNC identifier ( parameter_list ) statements
```

What might the C function that is responsible for parsing a function definition look like? It expects to first find the token FUNC, then it expects an identifier (the name of the function), followed by an opening parenthesis, and so on. As it pulls each token from the parser, it must ensure that it matches the expected, and if not, will halt with an error. For each nonterminal, this function calls the associated function to handle its part of the parsing. Check this out:

```
void ParseFunction()
{
   if (lookahead != T_FUNC) { // anything not FUNC here is wrong
      printf("syntax error \n");
      exit(0);
   } else
      lookahead = yylex();   // global 'lookahead' holds next token
   ParseIdentifier();
   if (lookahead != T_LPAREN) {
      printf("syntax error \n");
      exit(0);
   } else
      lookahead = yylex();
   ParseParameterList();
   if (lookahead!= T_RPAREN) {
      printf("syntax error \n");
      exit(0);
   } else
      lookahead = yylex();
   ParseStatements();
}
```

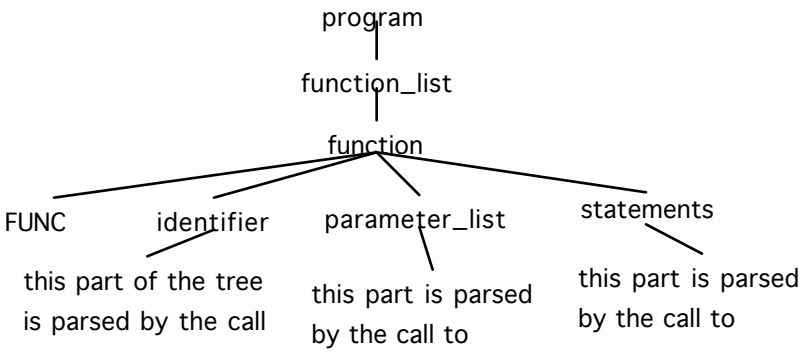
To make things a little cleaner, let's introduce a utility function that can be used to verify that the next token is what is expected and will error and exit otherwise. We will need this again and again in writing the parsing routines.

```
void MatchToken(int expected)
{
   if (lookahead != expected) {
      printf("syntax error, expected %d, got %d\n", expected,lookahead);
      exit(0);
   } else      // if match, consume token and move on
      lookahead = yylex();
}
```

Now we can tidy up the **`ParseFunction`** routine and make it clearer what it does:

```
void ParseFunction()
{
   MatchToken(T_FUNC);
   ParseIdentifier();
   MatchToken(T_LPAREN);
   ParseParameterList();
   MatchToken(T_RPAREN);
   ParseStatements();
}
```

The following diagram illustrates how the parse tree is built:



Here is the production for an if-statement in this language:

```
if_statement ->  IF expression THEN statement ENDIF |
                 IF expression THEN statement ELSE statement ENDIF
```

To prepare this grammar for recursive-descent, we must left-factor to share the common parts:

```
if_statement -> IF expression THEN statement close_if
   close_if   -> ENDIF | ELSE statement ENDIF
```

Now, let's look at the recursive-descent functions to parse an if statement:

```
void ParseIfStatement()
{
   MatchToken(T_IF);
   ParseExpression();
   MatchToken(T_THEN);
   ParseStatement();
   ParseCloseIf();
}

void ParseCloseIf()
{
   if (lookahead == T_ENDIF)  // if we immediately find ENDIF
      lookahead = yylex();    // predict close_if -> ENDIF
   else {
      MatchToken(T_ELSE);     // otherwise we look for ELSE
      ParseStatement();       // predict close_if -> ELSE stmt ENDIF
      MatchToken(T_ENDIF);
   }
}
```

When parsing the closing portion of the if, we have to decide which of the two right-hand side options to expand. In this case, it isn't too difficult. We try to match the first token again ENDIF and on non-match, we try to match the ELSE clause and if that doesn't match, it will report an error.

Navigating through two choices seemed simple enough, however, what happens where we have many alternatives on the right side?

```
statement  -> assg_statement | return_statement | print_statement | null_statement
                | if_statement | while_statement | block_of_statements
```

When implementing the **`ParseStatement`** function, how are we going to be able to determine which of the seven options to match for any given input? Remember, we are trying to do this without backtracking, and just one token of lookahead, so we have to be able to make immediate decision with minimal information— this can be a challenge!

To understand how to recognize and solve problem, we need a definition:

> The *first set* of a sequence of symbols $\underline{u}$, written as First($\underline{u}$) is the set of terminals which start the sequences of symbols derivable from $\underline{u}$. A bit more formally, consider all strings derivable from $\underline{u}$. If $\underline{u} \Rightarrow^* \underline{v}$, where $\underline{v}$ begins with some terminal, that terminal is in First($\underline{u}$). If $\underline{u} \Rightarrow^* \varepsilon$, then ε is in First($\underline{u}$).

Informally, the first set of a sequence is a list of all the possible terminals that could start a string derived from that sequence. We will work an example of calculating the first sets a bit later. For now, just keep in mind the intuitive meaning. Finding our lookahead token in one of the first sets of the possible expansions tells us that is the path to follow.

Given a production with a number of alternatives: A -> $\underline{u}_1$ | $\underline{u}_2$ | ..., we can write a recursive-descent routine only if all the sets First($\underline{u}_i$) are disjoint. The general form of such a routine would be:

```
void ParseA()
{
   // case below not quite legal C, need to list symbols individually
   switch (lookahead) {
      case First(u1):      // predict production A -> u1
         /* code to recognize u1 */
         return;
      case First(u2):      // predict production A -> u2
         /* code to recognize u2 */
         return;
        ....
      default:
         printf("syntax error \n");
         exit(0);
   }
}
```

If the first sets of the various productions for a nonterminal are not disjoint, a predictive parser doesn't know which choice to make. We would either need to re-write the grammar or use a different parsing technique for this nonterminal. For programming

languages, it is usually possible to re-structure the productions or embed certain rules into the parser to resolve conflicts, but this constraint is one of the weaknesses of the top-down non-backtracking approach.

It is a bit trickier if the nonterminal we are trying to recognize is *nullable*. A nonterminal A is nullable if there is a derivation of A that results in ε (i.e., that nonterminal would completely disappear in the parse string) i.e., ε ∈ First(A). In this case A could be replaced by nothing and the next token would be the first token of the symbol following A in the sentence being parsed. Thus if A is nullable, our predictive parser also needs to consider the possibility that the path to choose is the one corresponding to A =>* ε. To deal with this we define the following:

> The *follow set* of a nonterminal A is the set of terminal symbols that can appear immediately to the right of A in a valid sentence. A bit more formally, for every valid sentence S =>* $\underline{u}A\underline{v}$, where $\underline{v}$ begins with some terminal, and that terminal is in Follow(A).

Informally, you can think about the follow set like this: A can appear in various places within a valid sentence. The follow set describes what terminals could have followed the sentential form that was expanded from A. We will detail how to calculate the follow set a bit later. For now, realize follow sets are useful because they define the right context consistent with a given nonterminal and provide the lookahead that might signal a nullable nonterminal should be expanded to ε.

With these two definitions, we can now generalize how to handle A -> $\underline{u}_1$ | $\underline{u}_2$ | ..., in a recursive-descent parser. In all situations, we need a case to handle each member in First($\underline{u}_i$). In addition if there is a derivation from any $\underline{u}_i$ that could yield ε (i.e. if it is nullable) then we also need to handle the members in Follow(A).

```
void ParseA()
{
   switch (lookahead)
   {
      case First(u1):
         /* code to recognize u1 */
         return;
      case First(u2):
         /* code to recognize u2 */
         return;
      ...
      case Follow(A): // predict production A->epsilon if A is nullable
         /* usually do nothing here */
      default:
         printf("syntax error \n");
         exit(0);
   }
}
```

What about left-recursive productions? Now we see why these are such a problem in a predictive parser. Consider this left-recursive production that matches a list of one or more functions.

```
function_list       ->    function_list function | function
function            ->    FUNC identifier ( parameter_list ) statement

void ParseFunctionList()
{
   ParseFunctionList();
   ParseFunction();
}
```

Such a production will send a recursive-descent parser into an infinite loop! We need to remove the left-recursion in order to be able to write the parsing function for a function_list.

```
function_list    ->    function_list function | function
```

becomes

```
function_list    ->    function function_list | function
```

then we must left-factor the common parts

```
function_list    ->    function more_functions
more_functions   ->    function more_functions | ε
```

And now the parsing function looks like this:

```
void ParseFunctionList()
{
   ParseFunction();
   ParseMoreFunctions(); // may be empty (i.e. expand to epsilon)
}
```

Computing first and follow

These are the algorithms used to compute the first and follow sets:

Calculating first sets. To calculate First(u) where u has the form $X_1X_2...X_n$, do the following:

a) If X_1 is a terminal, add X_1 to First(u) and you're finished.

b) Else X_1 is a nonterminal, so add First(X_1) - ε to First(u).

a. If X_1 is a nullable nonterminal, i.e., $X_1 =>^* \varepsilon$, add First(X_2) - ε to First(u). Furthermore, if X_2 can also go to ε, then add First(X_3) - ε and so on, through all X_n until the first non-nullable symbol is encountered.

b. If $X_1X_2...X_n =>^* \varepsilon$, add ε to the first set.

Calculating follow sets. For each nonterminal in the grammar, do the following:

1. Place EOF in Follow(S) where S is the start symbol and EOF is the input's right endmarker. The endmarker might be end of file, newline, or a special symbol, whatever is the expected end of input indication for this grammar. We will typically use $ as the endmarker.
2. For every production A -> uBv where u and v are any string of grammar symbols and B is a nonterminal, everything in First(v) except ε is placed in Follow(B).
3. For every production A -> uB, or a production A -> uBv where First(v) contains ε (i.e. v is nullable), then everything in Follow(A) is added to Follow(B).

Here is a complete example of first and follow set computation, starting with this grammar:

```
S  ->  AB
A  ->  Ca | ε
B  ->  BaAC | c
C  ->  b | ε
```

Notice we have a left-recursive production that must be fixed if we are to use LL(1) parsing:

```
B  -> BaAC | c     becomes   B   ->  cB'
                             B'  ->  aACB' | ε
```

The new grammar is:

```
S  ->  AB
A  ->  Ca | ε
B  ->  cB'
B' ->  aACB' | ε
C  ->  b | ε
```

It helps to first compute the nullable set (i.e., those nonterminals X that X =>* ε), since you need to refer to the nullable status of various nonterminals when computing the first and follow sets:

```
Nullable(G) =  {A  B' C}
```

The first sets for each nonterminal are:

```
First(C) = {b ε}
First(B') = {a ε}
First(B) = {c}
First(A) = {b a ε}
    Start with First(C) - ε, add a (since C is nullable) and ε (since A itself is nullable)
First(S) = {b a c}
```

Start with First(A) - ε, add First(B) (since A is nullable). We don't add ε (since S itself is not-nullable— A can go away, but B cannot)

It is usually convenient to compute the first sets for the nonterminals that appear toward the bottom of the parse tree and work your way upward since the nonterminals toward the top may need to incorporate the first sets of the terminals that appear beneath them in the tree.

To compute the follow sets, take each nonterminal and go through all the right-side productions that the nonterminal is in, matching to the steps given earlier:

Follow(S) = {$}
S doesn't appear in the right hand side of any productions. We put $ in the follow set because S is the start symbol.

Follow(B) = {$}
B appears on the right hand side of the S –> AB production. Its follow set is the same as S.

Follow(B') = {$}
B' appears on the right hand side of two productions. The B' –> aACB' production tells us its follow set includes the follow set of B', which is tautological. From B –> cB', we learn its follow set is the same as B.

Follow(C) = {a $}
C appears in the right hand side of two productions. The production A –> Ca tells us a is in the follow set. From B' –> aACB' , we add the First(B') which is just a again. Because B' is nullable, we must also add Follow(B') which is $.

Follow(A) = {c b a $}
A appears in the right hand side of two productions. From S –> AB we add First(B) which is just c. B is not nullable. From B' –> aACB' , we add First(C) which is b. Since C is nullable, so we also include First(B') which is a. B' is also nullable, so we include Follow(B') which adds $.

It can be convenient to compute the follows sets for the nonterminals that appear toward the top of the parse tree and work your way down, but sometimes you have to circle around computing the follow sets of other nonterminals in order to complete the one you're on.

The calculation of the first and follow sets follow mechanical algorithms, but it is very easy to get tripped up in the details and make mistakes even when you know the rules. Be careful!

Table-driven LL(1) Parsing

In a recursive-descent parser, the production information is embedded in the individual parse functions for each nonterminal and the run-time execution stack is keeping track of our progress through the parse. There is another method for implementing a predictive parser that uses a table to store that production along with an explicit stack to keep track of where we are in the parse.

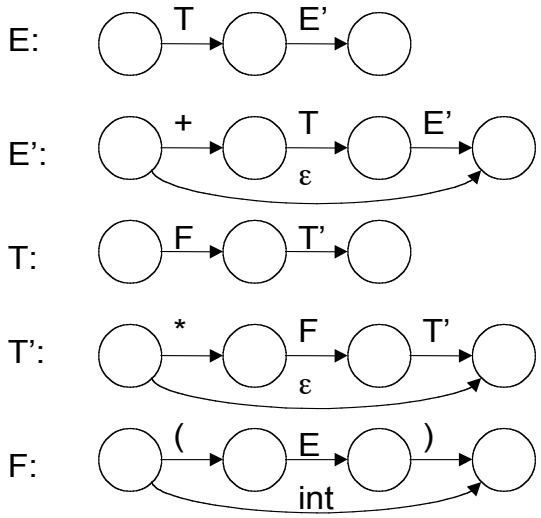
This grammar for add/multiply expressions is already set up to handle precedence and associativity:

```
E   ->   E + T | T
T   ->   T * F | F
F   ->   (E) | int
```

After removal of left recursion, we get:

```
E   ->   TE'
E'  ->   + TE' | ε
T   ->   FT'
T'  ->   * FT' | ε
F   ->   (E) | int
```

One way to illustrate the process is to study some transition graphs that represent the grammar:



A predictive parser behaves as follows. Let's assume the input string is `3 + 4 * 5`. Parsing begins in the start state of the symbol `E` and moves to the next state. This transition is marked with a `T`, which sends us to the start state for `T`. This in turn, sends us to the start state for `F`. `F` has only terminals, so we read a token from the input string. It must either be an open parenthesis or an integer in order for this parse to be valid. We consume the integer token, and thus we have hit a final state in the `F` transition

diagram, so we return to where we came from which is the T diagram; we have just finished processing the F nonterminal. We continue with T', and go to that start state. The current lookahead is + which doesn't match the * required by the first production, but + is in the follow set for T' so we match the second production which allows T' to disappear entirely. We finish T' and return to T, where we are also in a final state. We return to the E diagram where we have just finished processing the T. We move on to E', and so on.

A table-driven predictive parser uses a stack to store the productions to which it must return. A parsing table stores the actions the parser should take based on the input token and what value is on top of the stack. $ is the end of input symbol.

Input/ Top of parse stack	int	+	*	(	)	$
E	E–> TE'			E–> TE'		
E'		E'–> +TE'			E'–> ε	E'–> ε
T	T–> FT'			T–> FT'		
T'		T'–>ε	T'–> *FT'		T'–> ε	T'–> ε
F	F–> int			F–> (E)		

Tracing

Here is how a predictive parser works. We push the start symbol on the stack and read the first input token. As the parser works through the input, there are the following possibilities for the top stack symbol X and the input token a using table M:

1. If X = a and a = end of input ($), parser halts and parse completed successfully.
2. If X = a and a != $, successful match, pop X and advance to next input token. This is called a *match* action.
3. If X != a and X is a nonterminal, pop X and consult table at M[X,a] to see which production applies, push right side of production on stack. This is called a *predict* action.
4. If none of the preceding cases applies or the table entry from step 3 is blank, there has been a parse error.

Here is an example parse of the string int + int * int:

PARSE STACK	REMAINING INPUT	PARSER ACTION
E$	int + int * int$	Predict E –> TE', pop E from stack, push TE', no change in input

TE'$	int + int * int$	Predict T–> FT'
FT'E'$	int + int * int$	Predict F–> int
intT'E'$	int + int * int$	Match int, pop from stack, move ahead in input
T'E'$	+ int * int$	Predict T'–>ε
E'$	+ int * int$	Predict E'–> +TE'
+TE'$	+ int * int$	Match +, pop
TE'$	int * int$	Predict T–>FT'
FT'E'$	int * int$	Predict F–> int
intT'E'$	int * int$	Match int, pop
T'E'$	* int$	Predict T'–> *FT'
*FT'E'$	* int$	Match *, pop
FT'E'$	int$	Predict F–> int
intT'E'$	int$	Match int, pop
T'E'$	$	Predict T'–> ε
E'$	$	Predict E'–> ε
$	$	Match $, pop, success!

Suppose, instead, that we were trying to parse the input +$. The first step of the parse would give an error because there is no entry at M[E, +].

Constructing The Parse Table

The next task is to figure out how we built the table. The construction of the table is somewhat involved and tedious (the perfect task for a computer, but error-prone for humans). The first thing we need to do is compute the first and follow sets for the grammar:

```
E  ->  TE'
E' ->  + TE' | ε
T  ->  FT'
T' ->  * FT' | ε
F  ->  (E) | int

First(E) = First(T) = First(F) = { ( int }
First(T') = { * ε }
First(E') = { + ε }
Follow(E) = Follow(E') { $ ) }
Follow(T) = Follow(T') = { + $ ) }
Follow(F) = { * + $ ) }
```

Once we have the first and follow sets, we build a table M with the leftmost column labeled with all the nonterminals in the grammar, and the top row labeled with all the terminals in the grammar, along with $. The following algorithm fills in the table cells:

1. For each production A –> u of the grammar, do steps 2 and 3
2. For each terminal a in First(u), add A –> u to M[A,a]
3. If ε in First(u), (i.e. A is nullable) add A –> u to M[A,b] for each terminal b in Follow(A), If ε in First(u), and $ is in Follow(A), add A –> u to M[A,$]
4. All undefined entries are errors

The concept used here is to consider a production A –> u with a in First(u). The parser should expand A to u when the current input symbol is a. It's a little trickier when u = ε or u =>* ε. In this case, we should expand A to u if the current input symbol is in Follow(A), or if the $ at the end of the input has been reached, and $ is in Follow(A).

If the procedure ever tries to fill in an entry of the table that already has a non-error entry, the procedure fails— the grammar is not LL(1).

LL(1) Grammars: Properties

These predictive top-down techniques (either recursive-descent or table-driven) require a grammar that is LL(1). One fully-general way to determine if a grammar is LL(1) is to build the table and see if you have conflicts. In some cases, you will be able to determine that a grammar is or isn't LL(1) via a shortcut (such as identifying obvious left-factors). To give a formal statement of what is required for a grammar to be LL(1):

- No ambiguity
- No left recursion
- A grammar G is LL(1) iff whenever A –> u | v are two distinct productions of G, the following conditions hold:
  - for no terminal a do both u and v derive strings beginning with a (i.e., first sets are disjoint)
  - at most one of u and v can derive the empty string
  - if v =>* ε then u does not derive any string beginning with a terminal in Follow(A) (i.e., first and follow must be disjoint if nullable)

All of this translates intuitively that when trying to recognize A, the parser must be able to examine just one input symbol of lookahead and uniquely determine which production to use.

Error Detection and Recovery

A few general principles apply to errors found regardless of parsing technique being used:

- A parser should try to determine that an error has occurred as soon as possible. Waiting too long before declaring an error can cause the parser to lose the actual location of the error.

- A suitable and comprehensive message should be reported. “Missing semicolon on line 36” is helpful, “unable to shift in state 425” is not.
- After an error has occurred, the parser must pick a reasonable place to resume the parse. Rather than giving up at the first problem, a parser should always try to parse as much of the code as possible in order to find as many real errors as possible during a single run.
- A parser should avoid *cascading errors*, which is when one error generates a lengthy sequence of spurious error messages.

Recognizing that the input is not syntactically valid can be relatively straightforward. An error is detected in predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol and the parsing table entry M[A,a] is empty.

Deciding how to handle the error is bit more complicated. By inserting specific error actions into the empty slots of the table, you can determine how a predictive parser will handle a given error condition. At the least, you can provide a precise error message that describes the mismatch between expected and found.

Recovering from errors and being able to resume and successfully parse is more difficult. The entire compilation could be aborted on the first error, but most users would like to find out more than one error per compilation. The problem is how to fix the error in some way to allow parsing to continue.

Many errors are relatively minor and involve syntactic violations for which the parser has a correction that it believes is likely to be what the programmer intended. For example, a missing semicolon at the end of the line or a misspelled keyword can usually be recognized. For many minor errors, the parser can "fix" the program by guessing at what was intended and reporting a warning, but allowing compilation to proceed unhindered. The parser might skip what appears to be an erroneous token in the input or insert a necessary, but missing, token or change a token into the one expected (substituting BEGIN for BGEIN). For more major or complex errors, the parser may have no reliable correction. The parser will attempt to continue but will probably have to skip over part of the input or take some other exceptional action to do so.

Panic-mode error recovery is a simple technique that just bails out of the current construct, looking for a safe symbol at which to restart parsing. The parser just discards input tokens until it finds what is called a *synchronizing* token. The set of synchronizing tokens are those that we believe confirm the end of the invalid statement and allow us to pick up at the next piece of code. For a nonterminal A, we could place all the symbols in Follow(A) into its synchronizing set. If A is the nonterminal for a variable declaration and the garbled input is something like **`duoble d;`** the parser might skip ahead to the semi-colon and act as though the declaration didn’t exist. This will surely cause some more

cascading errors when the variable is later used, but it might get through the trouble spot. We could also use the symbols in First(A) as a synchronizing set for re-starting the parse of A. This would allow input `junk double d;` to parse as a valid variable declaration.

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

D. Cohen, Introduction to Computer Theory. New York: Wiley, 1986.

C. Fischer, R. LeBlanc, Crafting a Compiler. Menlo Park, CA: Benjamin/Cummings, 1988.

K. Loudon, Compiler Construction. Boston, MA: PWS, 1997.