Breadth-First Parsing

John Ophel

Department of Computing and
Electrical Engineering
Research Memorandum 97/12

Breadth-First Parsing

J. L. Ophel*
School of Computer Science
and Computer Engineering,
La Trobe University,
Bundoora, Victoria 3083,
Australia.
E-mail: `johno@lucinda.cs.latrobe.edu.au`

December 4, 1997

Abstract

Parsers perform three tasks: verify that their input is valid according to a grammar specification; provide a framework for associating semantic meaning with the input; and if the input is invalid, give an explanation of why the input is not valid. Current approaches to parsing usually require re-writing of an input grammar which complicates the association of meaning with the input. As well, additional mechanisms are required to provide error handling. Breadth-first parsing is a novel approach that performs traditional syntax checking and provides a framework for associating actions that is consistent with the grammar specification. It is also a simple but highly effective basis for error handling.

A breadth-first parser views a language grammar as a hierarchy of levels. Within each level the parser performs two passes. The first pass identifies the main constructs at this level by a simple, robust examination of the input. The second pass uses information from the first pass for syntax checking and breadth-first parsing of lower-level constructs. As the first pass identifies constructs that would normally require grammar re-writing, the second pass can be directly based on the original grammar and therefore provides a framework for associating actions with the input. The paper describes breadth-first parsing and demonstrates its application in a parser for the Pascal programming language.

1 Introduction

Many software systems use a parser to read input and check that it is valid according to a grammar specification[1]. Compilers are an obvious example, but tools to analyze source programs also use parsers, as can data manipulation programs. A parser must perform three tasks: verify that the input is valid; provide a framework for associating semantic meaning with the input[2]; and if the input is invalid, give an explanation of why and where the input is not valid.

There is a well-established theory and practice for developing parsers, either by hand or using a parser generator, that verify the validity of input [3, 1]. Unfortunately, the other tasks of a parser are less well supported[2].

Systems using a parser perform actions based on the meaning of the input which is derived from its syntactic structure. For example, a parser in a compiler may be annotated with actions that directly translate the source to a target based on the language semantics. The parser must therefore provide a framework for associating semantic actions. However, as most practical parsers are $LL(1)$ or $LR(1)$, grammars must usually be re-written to make them suitable for these parsers. A parser or parser specification based on a re-written grammar obscures the relationship between a syntactic construct and its associated semantic construct. For example, if a grammar is re-written to remove left recursion to make it $LL(1)$, it is usually

*This work was completed while the author was visiting the Department of Computing and Electrical Engineering at Heriot-Watt University, Edinburgh.

very difficult to understand the relationship between the re-written grammar and the semantic constructs. The loss of the syntax-semantics relationship makes it difficult to associate actions with the structure of the parser or parser specification. This affects the original developer of the parser, later maintenance of the parser, and increasingly, developers wishing to use an existing parser in a new system.

If a program is invalid, the parser must give accurate, informative error messages indicating why the program is invalid. The unfortunate reality of current practice is that parsers give poor error messages and there are often many cascading, spurious errors reported due to previous errors. Current practice is hampered by the lack of simple schemes for robustly recovering from errors and by complications introduced from re-writing the grammar[4].

This paper describes an alternative approach to parsing. *Breadth-first parsing* views a grammar as a hierarchy of levels. At each level, the parser works in two phases. The first phase performs a light-weight parse that identifies the important top-level constructs of the input. As constructs are identified, the initial sequence of tokens is re-written as a tree with place-holder tokens. The second phase performs more traditional syntax checking, but as the first phase has identified constructs that would normally require grammar re-writing, the second phase can be directly based on the original grammar. The second phase also does breadth-first parsing on lower-level constructs.

The paper argues that breadth-first parsing has two main advantages over traditional approaches. First, common syntactic forms are parsed in a manner that can maintain the syntax-semantics relationship. Second, there is a simple error handling strategy: syntactic constructs that have been identified can be checked independently so errors in one construct are isolated from other constructs, making error recovery robust and reducing the production of spurious errors. Breadth-first parsing is more expensive than traditional depth-first approaches: it requires more than one pass over the input and uses more storage. The overheads of breath-first parsing are discussed below.

The paper has five main sections following this introduction. Section 2 describes breadth-first parsing and Section 3 gives an example using a small grammar. Section 4 then describes a breath-first parser for the language Pascal. The error handling in the parser is examined in Section 5, and some examples given. The Section 6 then discusses the overheads of breadth-first parsing. Finally a related work section compares this approach with previous work and the conclusions section discusses the contribution of breadth-first parsing and future work.

2 Breadth-First Parsing

Breadth-first parsing stratifies a grammar into a hierarchy of levels. Within each level the parse has two phases. The first, the *articulation* phase, identifies the constructs of the current level. When a constructs is identified, the start and end of the construct is determined, as well as what sort of construct it is. The second phase, *syntax checking*, then does the more traditional syntax checking and also does breadth-first parsing of the sub-constructs of the level below. For example, in a breadth-first parser for the language Pascal[5], the grammar would have three levels: program, statement and expression. The parsing at the program level would first articulate the source program so that all declarations and the main body of the program were identified. The second phase of the program-level parse would then check the syntax of each construct. The checking of subprograms and the main body would then in turn articulate statements before checking their syntax. The whole breadth-first parser is written as a recursive-descent parser with calls to articulate levels when appropriate.

The name "*breadth-first*" was chosen to emphasize that during articulation the input is examined but lower levels are not parsed, only the constructs of the current level are of interest. The articulation phase can be viewed as a light-weight parse that identifies constructs using a subset of actual input tokens. In this way it leads to a more hierarchical approach to parsing - first the overall structure is identified, then checked, and then sub-constructs identified, and so on.

As the input is examined more than once, input tokens are stored in an internal data structure. The articulation phase adds place-holder tokens to the initial sequence of input tokens to form a tree that represents the hierarchical structure of the input.

The articulation phase identifies constructs by using a subset of the input tokens called *feature tokens*. The feature tokens determine the start and end of the constructs in the current level. An underlying assumption of breadth-first parsing is that constructs at a level can be identified without fully parsing the input. If a feature token used to identify a construct can appear in other places in the grammar, then there must be some way to recognize when it is performing a distinguishing role. Programming language syntax typically use either brackets or pairs of symbols to nest or use feature tokens in a different context; for example, Pascal uses `begin/end` pairs to nest groups of statements. In a grammar with nested feature tokens within delimiting pairs, the articulation phase is able to keep track of nesting and therefore determine the end of the main construct.

The choice of levels reflects the hierarchical structure in the semantics denoted by the input grammar. Most programming languages have such structure as described for Pascal above with program-level constructs, statements and expressions.

The second phase syntax checking is traditional recursive-descent parsing, except that the input has place-holder tokens indicating identified constructs. As articulation can resolve issues that normally require a grammar to be re-written, the syntax checking can be based directly on a grammar that reflects semantic structure. It is therefore a relatively simple process to annotate the parser with semantic actions required by the application using the parser.

Breadth-first parsing provides a framework for developing robust error handling. In traditional depth-first parsers, when an error is detected, the parser must have a mechanism for recovering from the error so that it can detect later errors. Error recovery mechanisms are complex and subtle; spurious errors are often generated. The hierarchical nature of breadth-first parsing gives it a simple framework for error recovery. In a breadth-first parser, errors are detected both in the articulation and syntax checking phases. Articulation errors imply that the current construct being articulated has gross errors and parsing is stopped at this point. In the syntax checking phase, the error can be reported and the basic error recovery is to stop parsing this construct and continue at the next construct. More sophisticated error handling is possible, both during articulation and syntax checking, and is described in Section 4.

3 A Small Example

Consider the grammar[1] shown in Figure 1. The grammar describes a simple language of identifiers, arrays, and procedure call and assignment statements. Although the grammar is simple and intuitive, the process of writing the parser is problematic. The grammar causes difficulties for both *LL* and *LR* parsers as the input `a(i,j)` can be either a complete procedure call statement or an array access on the left-hand side of an assignment. An *LL* parser cannot decide which statement production to predict on reading the token `a`, an *LR* parser cannot decide whether to reduce the token `i` to an expression or a parameter.

```
statements -> statements ; statement
            | statement

statement -> id ( parameter-list )
           | expression := expression

parameter-list -> parameter-list , parameter
                | parameter

parameter -> id

expression -> id
            | id ( expression-list )

expression-list -> expression-list , expression
                 | expression
```

Figure 1. An example grammar.

One approach when writing a parser for such a grammar is to re-write the grammar. One solution would merge a *parameter* and an *expression*. Changing a grammar, however, introduces additional problems. Re-writing one part of a grammar often affects other parts of the grammar, leading to a ripple of changes. Checks must be provided after parsing to detect invalid input that is accepted by the re-written grammar. The correspondence between syntactic and semantic constructs is lost. For example, if the compiler is generating an abstract syntax tree to represent the program for later phases of compilation - what tree should be generated when `a(i,j)` is parsed? Finally, the quality of error reporting is reduced - if an error occurs in an *expression* it is reported as an *expression* error, even if it is really a *parameter* error.

Another approach to writing a parser for this grammar would perform semantic analysis, either in the scanner or the parser, to distinguish procedure call identifiers and array names. This approach is heavy-handed, complicates the parsing and results in obscure error messages; for example, if an identifier is undeclared.

A breadth-first parser for the above grammar would have two levels: the statement level and the base level. These levels reflect the hierarchy of the language grammar and semantics - a series of statements where each statement is an assignment or a procedure call. At the statement level the articulation phase would break the sequence of statements into individual statements and distinguish each particular statement. This is simple: a statement is a sequence of tokens terminated by a semi-colon or the end of input (end-of-file). If the sequence of tokens contains an assignment operator (`:=`) then the statement is an assignment statement. If the tokens are an identifier and brackets, then the statement is a procedure call. Otherwise, it is a badly formed statement.

For the input `a(i,j); a(i,j) := x` the first pass of the breadth-first parser would effectively produce a representation in the following form:

```
procedure-call statement -- ; -- assignment statement
           |                                |
           |                                |
         a(i,j)                       a(i,j) := x
```

The initial list of tokens has been transformed into a tree by the addition of the *procedure-call statement* and *assignment statement* place-holder tokens. The second phase of the statement level parse would check the syntax, that is, that there is a sequence of statements separated by semi-colons. As each statement was checked, it would be parsed at its base level. As the sort of statement is determined by the first phase, the second phase syntax checking can be directly based on the initial grammar.

As an example of the robustness of breadth-first parsing, the invalid input `a(i,,j); a() := x` has enough structure so that articulation would produce the tree:

```
procedure-call statement -- ; -- assignment statement
           |                                |
           |                                |
        a(i,,j)                         a() := x
```

The error in each statement can be handled independently and reported in its appropriate context.

For the invalid input `a(i,j) := y b(i,j) := x` the articulation phase can report that the overall structure is invalid as two assignment statements are not separated by a semi-colon. Most traditional parsing approaches would recognize an error on the `b` token but would treat the error as a invalid expression rather than the more global view of the missing semi-colon.

For the invalid input `a(i,j) b(i,j)` the articulation phase would place all tokens into one statement. When trying to recognize the sort of statement, an error would occur as this is a badly formed statement.

The articulation phase relies on the semi-colon feature token only appearing as a statement separator. What if semi-colons could appear inside expressions? The grammar might have the additional production:

```
expression -> ( expression ; expression )
```

Such a construct would evaluate each expression and return the value of the second expression as the value of the whole expression. Individual *statement*s can still be articulated by the semi-colon - all semi-colons within parentheses are ignored. If a semi-colon appears after an unclosed left parenthesis, the semi-colon is not a feature token and is therefore not a statement separator.

If the grammar did not require a semi-colon separator between statements, then the breadth-first parser would not be able to identify statements without parsing the input. The breadth-first parser would then have only one level and would be equivalent to a traditional recursive-descent parser.

4 A Breadth-First Parser for Pascal

This section describes the design and implementation of a breadth-first parser for Pascal[5]. Pascal was chosen as a vehicle for demonstrating the ideas of breadth-first parsing as it is well-known and large enough as a real test of ideas. Discussion of error handling is deferred until the following section.

The language is stratified into three levels: the program, statement and expression levels. The program level consists of declarations and the main body of the program. Declarations include label, constant, type and variable declarations as well as procedure and function declarations and definitions. A subprogram declaration (where a `forward` directive appears) is distinguished from a definition (where a body is defined) to emphasize the different semantic entities.

All input tokens are initially placed into a linked list to simplify the interaction between the scanner and the parser; an alternative implementation could have the top-level articulation phase calling the scanner directly. The program-level articulation phase takes a linked list of tokens and transforms this list into a tree by adding place-holder tokens to represent the various declarations and the main body. As subprogram definitions cannot be identified without identifying nested subprogram definitions, all local declarations and the body of a subprogram definition are identified during the program-level articulation. The articulation phase also adds a program declaration place-holder as the root node of the tree.

The articulation phase of the Pascal parser uses *feature* tokens. The feature tokens are a subset of the input tokens and consist of the tokens that identify program-level constructs. For the program level these include `program`, `label`, `const`, `type`, `var`, `procedure`, `function`, `forward`, `begin` and `end`. Tokens such as semi-colon and the final dot of the Pascal program are not used as feature tokens as they are used in many constructs and are not necessary to perform the articulation. As subprograms and the main body of a Pascal program cannot be identified unless all pairs of `begin` and `end` tokens are matched, both `record` and `case` are added to the feature tokens as they match `end` tokens as well. The tokens `var`, `procedure` and `function` can appear in the formal parameter section of a subprogram declaration or definition. However, a formal parameter section has nesting parentheses, so these are also added to the feature tokens. Finally both start of file and end of file tokens are added to the feature set to simplify boundary cases.

A stack based transformer was used during articulation. Two stacks are used: one a stack of feature tokens and constructs (such as label declaration) and a stack of constructs. The feature stack guides the articulation process and the construct stack keeps track of currently open constructs. The constructs need not appear on the feature stack but it simplifies the process as described below.

Articulation traverses the linked list of tokens, skipping non-feature tokens. On encountering a feature token, a series of actions are performed based on the token and the feature stack. For example, when a `label` token is seen, then (using liberal Pascal syntax that allows declarations in any order) if there is a currently open declaration, it is closed. If a label declaration is valid at this point (a program declaration representing the whole program is currently open), a label declaration is opened by inserting a place-holder token and attaching the `label` token token as a child of the place holder. Both the label declaration place holder and the `label` token are pushed onto the feature stack.

In most cases, decisions can be made by examining the top of the feature stack and the current token. For example, when the current token is a left parenthesis and the top of the stack is a `procedure` token, the left parenthesis is the start of the procedure's formal parameter section. A parentheses construct is started and the construct and the actual token are pushed onto the feature stack. If the next feature token is a `var` token, then as the top of the feature stack was a left parenthesis, the `var` would be treated as a non-feature token as it is not an indicator of a variable declaration construct when nested in a formal parameter section.

In some cases, more than just the top token of the feature stack must be examined. For example, when the `begin` and `end` of a procedure definition are seen, the procedure definition is not actually complete - the Pascal syntax includes a semi-colon as the last token of a subprogram definition. The feature stack then has the open procedure definition construct, the `procedure` token and the `begin` and `end` tokens on the stacks, with the `end` token on top of the stack. When the next feature token is encountered, say a `begin` token, then the `end` on the stack may be part of a function definition or an entire program declaration. Thus more tokens on the feature stack must be checked to distinguish these cases.

Most program-level constructs can be identified by their first token - for example a label declaration by its `label` token. Procedure declarations and definitions are not identified until later tokens are seen. A declaration is identified by the occurrence of a `forward` token, a definition by a `begin` token signifying a procedure body. In such cases a temporary place holder is used. When later tokens allow the place holder to be fully identified, it is changed.

The articulation will therefore treat a procedure heading followed by local declarations followed by a `forward` directive as a procedure declaration. The syntax check for the procedure declaration will report that the local declarations are invalid. An alternative approach might use the appearance of the local declarations as an indication of a procedure definition and then the articulation would report an error at the `forward` directive.

As subprograms cannot be identified without any inner subprograms also being identified, the top-level articulation identifies all declarations and subprograms declared within a subprogram.

The full decision table for the program-level articulation is given as Appendix 1.

An example of an articulation is given in Figure 2. The source program is shown, along with the states of the feature stack as feature tokens are processed. For example, the first feature token is `start-of-file` which is pushed onto the stack to initialize the articulation. The next feature, the `program` token, starts a new construct, a program declaration, and pushes the `program` token. Starting a new construct pushes the construct on both the feature and construct stack. It also changes the point that tokens are added to the tree. The next token `p` is not a feature token so is added to the tree as a child of the program declaration construct. When the `procedure` token is processed, it first ends the variable declaration. The construct stack indicates what is popped off the feature stack (both the variable declaration construct and the `var` token). The new construct is not fully determined by the `procedure` token alone. An undetermined declaration is therefore started which is changed later when the body of the procedure definition is seen. The result of articulation is shown in Figure 3.

```
start-of-file
program p();
var
  x: real;
procedure p
   (var a: real);
begin
  if a > 10 then
  begin
    a := 10
  end
end
begin
  a := 5;
  p(x)
end.
```

feature stack	next token
	sof
sof	program
sof prog-decl program	var
sof prog-decl program var-decl var	procedure
sof prog-decl program ??-decl procedure	(
sof prog-decl program ??-decl procedure parentheses (	var
sof prog-decl program ??-decl procedure parentheses (	)
sof prog-decl program ??-decl procedure	begin
sof prog-decl program proc-defn procedure begin-end begin	begin
sof prog-decl program proc-defn procedure begin-end begin begin	end
sof prog-decl program proc-defn procedure begin-end begin	end
sof prog-decl program proc-defn procedure begin end	begin
sof prog-decl program begin-end begin	end
sof prog-decl program begin end	eof

Figure 2. An example of a program-level articulation.

When program-level articulation is complete, syntax checking at the program level is performed. The syntax checking is similar to a traditional recursive descent parse except it is performed on a tree of tokens, with place holders guiding the parse. For example, when the syntax of the program declaration is checked for the previous example, the parser will see the tokens `program`, `p`, `(`, `)` and `;` before place-holder tokens for a variable declaration, a procedure definition and a main body followed by the `.` token.

The statement-level articulation is performed when the program-level syntax checker is dealing with subprogram definitions and the main body of the program. Although articulation is based on the same

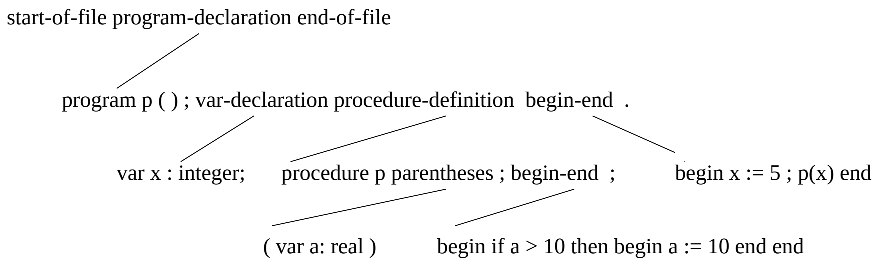

Figure 3. Tree of tokens after program-level articulation.

stack-based recognizer, there are some interesting differences.

First, statements may include other statements. For example, a while statement includes a following statement. The following statement may be made of other statements in turn. A repeat statement can contain an arbitrary number of statements. The articulator has an action to close a series of statements up to some point on the stack. For example, an `until` token requires that all open statement constructs up to its opening `repeat` are closed. The actual repeat statement is not closed at this point as there are more tokens in the statement after the `until` (the iterating expression).

The starting token of some statements is not a statement-level feature token. For example, an assignment statement has only one feature token (the `:=` token) which appears inside the statement. Statements are therefore indicated by a preceding token: `;`, `begin`, `do`, `then`, `else`, `:` (for a labelled statement or choice in a case statement) or `repeat`. When one of these tokens is encountered, an undetermined statement construct is opened and pushed on the feature stack. When an identifying feature token (or tokens) is encountered, the construct is changed to the appropriate sort of statement. Breadth-first parsing is based on the assumption that the start of a construct can be identified by either a preceding feature token or the first token of the construct. If this is not true, then either the levels of the parser must be altered or the set of feature tokens must be expanded.

Neither a null statement nor a procedure call statement contain feature tokens. When an open statement construct is closed (by one of the closing tokens `end`, `;`, `until` or `else`) then if the statement is still undetermined it must be identified. If the undetermined statement place holder has no child token then it is a null statement. If it is an identifier or an identifier with a bracketted sequence of tokens, it is a procedure call. Otherwise the statement is mal-formed.

The full decision table for statement-level articulation is given as Appendix 2.

As the expression level is the bottom level in the hierarchy of levels and it can be parsed directly, it is not articulated.

5 Error handling

A key component of a parser is its error handling capabilities. Error handling incorporates two activities: error reporting and error recovery[6]. Error reporting identifies the actual error that has been detected and produces a message. Error recovery must then attempt to decide how to continue the parse.

Traditional parsers typically have inconsistent error reporting and handling. Some errors are handled very well while others produce obscure error messages and the recovery is poor, so that a cascade of spurious errors is reported interleaved with other errors in the program. When compiling, many programmers simply fix the first reported syntax error and re-compile the fixed source, rather than attempt to wade through possibly spurious errors. Most of the spurious errors are produced because error recovery must either attempt to repair the input so that the parse can continue or find some point to re-synchronize the parse, hoping that the re-synchronization point is correctly chosen.

As described in the overview, a breadth-first parser identifies errors either during articulation or during syntax checking. The basic framework of the error recovery is to stop parsing the particular construct in error but continue with other constructs. Error recovery does not have to attempt to repair the input and has synchronization points that have been identified during previous articulation, making this framework more robust than previous approaches.

When an articulation error occurs, the point of detection is a feature token that may be many (non-feature) tokens past the actual error. The error handling uses the feature stack and the current token to report and recover from the error. As with articulation, in most cases the top of the feature stack is sufficient, but in other cases more of the feature stack is examined. The error reporting does not attempt to pinpoint the "exact" site of the error, as many errors have alternative causes. Instead the error handler can give an indication of the group of tokens in error.

When reporting errors in the syntax checking phase, a breadth-first parser can give a clear explanation of the construct that is in error because previous articulation has identified the construct. So even in a simple error handling scheme, rather than simply stating that an error occurred at a certain token, the message can refer to the enclosing identified construct.

There are situations that must be carefully dealt with by a breadth-first parser. The syntax checker must be aware that incorrect input may lead to constructs not being correctly articulated. This occurs when all the feature tokens of a construct are missing or misspelt, or if a construct has no feature tokens and its closing feature token is missing. In such cases the construct is not articulated, and it typically is merged with another construct. All syntax checking must therefore check that there are no extra tokens in this construct and if there are, that it may be due to missing tokens that prevented an accurate articulation.

The basic framework of error recovery in a breadth-first parser can be extended. This has been done in the Pascal parser and is described below. The error handling for this parser is simple but performs favourably on a standard set of invalid Pascal programs[7].

5.1 Articulation Error Handling

The articulation error recovery is fairly simple and conservative. If there is a single token deletion or insertion, this is suggested. Deletion is decided by a test of evidence. For example, if `record` is on top of the feature stack and `var` is encountered, then an error is reported. Evidence for the `var` token is considered. If the (non-feature) tokens after the `var` do not look like a variable declaration then the `var` token is deleted. Otherwise, if the tokens after the `record` do not look like a record declaration then it would be deleted. If both have supporting evidence then a possible insertion is considered. In the above case, a missing `end` token to close the `record` would be considered. For such a missing token, a simple spelling check of preceding tokens is performed. If the insertion would not require other tokens later in the input, the articulation continues. For example, if the error recovery decides that a `procedure` token is missing to correct the appearance of a `label` token, then there is now an assumption that the procedure definition will have a body. This is a source of potential spurious errors - if the missing `procedure` is not the real cause of the error. In such a case, the articulation would stop. If more than one insertion or deletion is required, then recovery also stops the articulation.

5.2 Syntax-Checking Error Handling

During syntax checking, a simple scheme is used. If a particular token is expected but not encountered, and it is the first error at this level for this construct, an error is reported. Recovery is simple, the parse just continues. If another error is detected, then the error is not reported. Eventually the construct is finished. If the construct contains internal feature points then the error reporting flag can be reset and later errors reported. For example, for an *if-then-else statement* at most one error is reported in the conditional expression but as the `then` token must appear (as the articulation phase has determined the sort of statement and was valid) then possibly erroneous input can be read up to the `then` token and later errors reported.

The errors reported are simple: a particular token was expected but another was found. Such error messages could be improved by a more sophisticated scheme that used knowledge of expected tokens and the actual tokens to suggest if a token or tokens should be inserted, deleted or are incorrect.

To provide the context of the error message, the syntax checker maintains a stack of string descriptors for the context being parsed, along with the first token of each context. For example, when the procedure to parse an *if-then-else statement* is called, the string *"if-then-else statement"* is pushed onto the stack. Any errors can then include information on the enclosing context of the error.

The syntax checking must deal with cases where a missing token affects the articulation. For example, the Pascal input `p(3) x := 10` will be seen as one assignment statement by the statement-level articulation phase as the procedure call statement separator is missing. The syntax check of the assignment statement will then report an invalid variable error. If the input was `p(2) if x > 10 then x := 0;` then the tokens from `p` up to the semi-colon would be included as part of an *if-then* statement (which includes an *assignment* statement). As the *if-then* statement has a feature token as its first token, a special call that expects the feature token is used. If the feature token is not the first token, then the error message suggests that a separating token is probably the cause of the error. As these situations occur in well defined places, the syntax checker can deal with them appropriately and assist the error handling.

5.3 Some Examples

Three examples of error handling in a traditional Pascal parser[8] and the breadth-first parser are given in Figures 4, 5 and 6. The error messages of the breadth-first parser are shown in "raw" form and would be further processed in a working compiler - typically showing some listing of the source. Comparing any two parsers for error handling is difficult as usually errors are handled inconsistently and each parser has various strengths and weaknesses. The point of the examples, however, is to show that compared to a "real" parser with a complex error handling mechanism, the simple error handling scheme of breadth-first parsing gives good error messages while avoiding spurious and confusing messages.

In Figure 4, there are two errors. First, the closing `end` token of the record declaration is misspelt. Second the final `end.` of the program is missing. The traditional parser parses the invalid `endd` token as a field of the record and then gets very confused. Spurious error messages are generated, swamping the message for the second error. The breadth-first parser detects the first error when the `var` token is encountered. As both the `record` and `var` tokens have evidence of being locally correct, the error handling looks for the missing `end` that is correctly identified as the misspelt `endd` token. The second error is also reported.

In Figure 5, there is only one error. The name of the program has incorrectly used the reserved word `begin`. The traditional parser has attempted to fix the input so that the `begin` token is valid. As the parser does not have any notion of the overall structure of the program it cannot determine that the `begin` is invalid and again, spurious error messages are generated. The breadth-first parser is able to determine that the `begin` as the name of the program is an error and as the evidence (the tokens following the `begin`) do not support its appearance, it is deleted. The breadth-first parser is not however, able to report the exact cause of the error but instead can only give a general, but useful, message.

In Figure 6 there are three minor errors. The first and last are detected by the syntax checking phase, the other by the articulation phase. The malformed statement error in the articulation phase is considered sufficiently self-contained so that syntax checking of the rest of the statements is not stopped. The errors detected in the syntax checking phase include the context of the error as part of the error message. The first error is better reported by the traditional parser with a sophisticated error handler. The invalid statement is handled better by the breadth-first parser, the other parser sees the `a = 0` as an expression and attempts to build an *if-then statement*, even though this introduces spurious errors. The missing separator produces a poor message from the breadth-first parser - but at least it gives an indication of the problem and a context for the user to try to understand the error.

6 The Overheads of Breadth-First Parsing

Breadth-first parsing has two additional costs over traditional parsing. First, it works by articulating a data structure representing all tokens of the source language. This is a linked list originally and as each level is articulated becomes a more nested tree with place holders. Second, the parser traverses the tree of tokens more than once for each level that is articulated - once for articulation and again for syntax checking.

Comparing the actual performance of software tools is difficult. Simple comparisons between a Flex/Bison-generated Pascal parser and the breadth-first parser showed that the breadth-first parser was between 10% to 50% slower for valid Pascal programs up to 5,000 lines. Both parsers simply checked for syntactic correctness; unlike the breadth-first parser, the Bison-generated parser performed no error recovery (it stopped on the first error encountered).

Program

```
program P(input,output);
type
  r = record
    a : integer;
  endd;          (* <-- error, "endd" should be "end" *)
var x : integer;
procedure pp;
begin
  write(2)
end;
begin
                 (* <-- error, missing "end." *) \\
```

Traditional Pascal Compiler

```
     5    endd;
E ------------^--- Malformed record declaration
     6  var x : integer;
e ------^--- Replaced keyword var with a ';'
     7  procedure pp;
E ------^--- Expected keyword end
E ------^--- Inserted keyword end matching record on line 4
e ------^--- Inserted ';'
    11  begin
E -----------^--- Malformed declaration
  -----------^--- Unexpected end-of-file - QUIT
```

Breadth-First Parser

```
Error: encountered at var line 6, column 1.
Possible missing "end" token - is "endd", line 5, column 3, misspelt?

Error: encountered at <eof> line 13, column 1
Possible missing "end" token - inserted by parser.
```

Figure 4. First error handling example.

The breadth-first parser used more memory. The *LR* parser generated by Bison is very space efficient using approximately 100K bytes for internal tables, independent of the size of the input file. In contrast the breadth-first parser used approximately 600K bytes to store the tree of tokens in a 5,000 line program. The program had approximately 21,000 tokens and each token used 28 bytes.

The performance of the breadth-first parser is perhaps surprising. Traditionally, examining tokens more than once or storing tokens internally has been seen as too much of an overhead for a parser. Instead, these overheads seem relatively modest.

Traditional parsing techniques were designed when the speed of scanning and parsing were an important factor in the speed of a compiler. Similarly, memory use was expensive. Given that scanning and parsing are now not the dominant cost of compiling as more semantic analysis and better code generation are performed, and that memory is much more plentiful, the costs imposed by the breadth-first parser are small compared to the advantages.

Program

```
program begin();       (* <-- error, "begin" used as name of program *)
type
  t =
  record
    x : integer;
  end;
var
  x,y : integer;
begin
  if x > y then write(x)
end.
```

Traditional Pascal Compiler

```
     1  program begin();
E --------------^--- Expected identifier
e --------------^--- Inserted ';'
E -------------------^--- Inserted identifier
     3  type
E ------^--- Declaration found when statement expected
E ------^--- Declaration found when statement expected
```

Breadth-First Parser

```
Error: begin, line 1, column 9, appears out of place - deleting token.
```

Figure 5. Second error handling example.

7 Related Work

The field of parsing is very well established[3, 1]. Traditional parsing is based on the assumption of examining the input in a left-to-right manner, examining each token only once. Lookahead parsers ($LL(k)$[9, 10] and $LR(k)$[11] for $k > 1$) allow more than just the next token to be used to guide the parse.

Back-tracking parsers[12] allow a token to be examined more than once. At a decision point, where more than one alternative construct is possible, each alternative is tried in turn. If an alternative is not possible, any exploratory parsing done is undone (the parsing is back-tracked) and the next alternative tried. Back-tracking has traditionally been viewed as too expensive for practical parsers but current experience is more positive[2, 12]. Breadth-first parsing effectively does its exploratory parsing during articulation, but unlike back-track parsing, is more robust as lower-level syntactic errors do not affect the articulation. Also, breadth-first parsing can include ad-hoc mechanisms (such as the checks to distinguish between null, procedure call and malformed statements).

Tree re-writing methods have been used in a parser to provide a framework for associating semantic actions with a grammar with conflicts that make it non-LR(1)[2]. The user associates semantic actions with the original grammar and the tree re-writing algorithm solves parsing conflicts by re-writing the grammar. During the parse, information is maintained so that actions are performed in accordance with a parse tree based on the original grammar. Such an approach has a similar motivation to breadth-first parsing but is effectively hiding the grammar re-writing rather than using an articulation phase to avoid the re-writing. Also, the tree re-writing does not address error handling.

Articulation has been used by some Fortran compilers in an ad-hoc way to deal with difficult language constructs[13, 14]. Such systems have not used articulation through-out the parse in the consistent manner of breadth-first parsing.

Software tools such as the *ctags* utility use ideas similar to articulation to "parse" source programs[15, 16]. Such tools are often not interested in many constructs and instead focus on items of interest. The items

Program

```
program p(input, output);
var
  a integer;              (* <-- error, missing ":" before "integer" *)
begin
  a = 0;                  (* <-- error, invalid statement *)
  a = 1                   (* <-- errors, invalid statement and missing semi-colon *)
  if (a > 10) then a := 10
end.
```

Traditional Pascal Compiler

```
     3    a integer;
e ----------^--- Inserted ':'
     5    a = 0;
e --------^--- Inserted keyword if
E ------------^--- Missing/malformed expression
     6    a = 1
e --------^--- Inserted keyword if
     7    if (a > 10) then a := 10
e --------^--- Inserted keyword then
```

Breadth-First Parser

```
Error: in variable declaration starting at line 2, column 1
line 3, column 5, expecting : but encountered integer
Error: malformed statement at a, line 5, column 3.
Error: in if-then statement starting at line 6, column 3
line 6, column 3; invalid first token - missing separator?
```

Figure 6. Third error handling example.

are feature tokens, often supplemented with non-grammatical heuristics (such as all function definitions are preceded by a blank line).

There is a large body of work in the field of error handling[6]. Most error recovery has concentrated on *local recovery* involving the insertion, deletion or replacement of one or more single tokens in the source. Less work has looked at *regional* or *global* recovery where larger sections of the source program are considered when decisions are made. The articulation phase of breadth-first parsing uses *regional* recovery as, although it is only considering two or three feature tokens, these tokens are markers for "phrase" level constructs. The evidence used in considering deletion when an articulation error occurs is similar to local recovery where an action is taken if the next couple of tokens are then valid. The error recovery during syntax checking is standard error handling for a recursive descent parser, with the additional advantage that since the feature points are known in advance, the error recovery can use these to continue parsing after an error has occurred.

One popular error recovery technique for recursive descent parsers is the use of *synchronization* tokens[17]. When parsing a particular construct, if an error occurs, tokens are skipped until a member of the set of synchronization tokens is encountered. A synchronization token is an "important" token, one that is a good place to try and continue parsing from. Articulation in breadth-first parsing effectively finds good synchronization points in a systematic way before doing a comprehensive parse.

8 Future Work and Conclusions

There remains much work to be done on breadth-first parsing. Breadth-first parsers need to be written for other languages to see if the ideas used in the Pascal parser are generally applicable or must be adapted or extended. The descriptions in this paper have been informal; a more formal description is required. Given a more formal description, the next stage of work is developing an automatic breadth-first parser generator.

There are several variations possible on the basic breath-first parser. One is an incremental version that articulates the next construct and then parses it fully before continuing. Another is a global error handling version that does no construct identification until all feature tokens are extracted. There is also potential parallelism after articulation where each construct can be separately checked.

Breadth-first parsing is a novel approach to parsing. By including an articulation phase and working in levels, a breadth-first parser can hierarchically decompose a program when parsing. This allows for more expressive parsers to be written that avoid the need for re-writing of grammars. The syntax checking parts of the parser more consistently follow the syntax to semantics mapping of the language. The extra information and decomposition provided by the articulation phase simplify the error handling and allow error reporting and recovery that is informative and useful, while avoiding cascading spurious errors.

9 Acknowledgements

Greg Michaelson, Norman Scaife and Tony Cox provided feed-back on an earlier draft of the paper. This work was completed while the author was visiting the Department of Computing and Electrical Engineering at Heriot-Watt University, Edinburgh.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, 1986.

[2] G. Harford, V. Heuring, and M. Main, ‘A new parsing method for non-LR(1) grammars’, *Software–Practice and Experience*, **22**(5), 419–437 (1992).

[3] A. Aho and J. Ullman, *Theory of parsing, translation and compiling*, volume 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[4] R. Backhouse, *Syntax of Programming Languages: Theory and Practice*, Prentice-Hall, Englewood, 1979.

[5] K. Jensen and N. Wirth, *PASCAL - User Manual and Report*, Springer-Verlag Lecture Notes in Computer Science 18, 1974.

[6] K. Hammond and V. J. Rayward-Smith, ‘A survey on syntactic error recovery and repair’, *Computer Languages*, **9**(1), 51–67 (1984).

[7] D. Ripley and F. Druseikis, ‘A statistical analysis of syntax errors’, *Computer Languages*, **3**(4), 227–240 (1978).

[8] Sun Pascal Compiler, ‘Sun Pascal User’s Guide’.

[9] T. Parr, H. Dietz, and W. Cohen, ‘PCCTS reference manual (version 1.00)’, *SIGPLAN Notices*, **27**(2), 88–165 (1992).

[10] T. Parr and R. Quong, ‘ANTLR: a predicated-LL(k) parser generator’, *Software–Practice and Experience*, **25**(7), 789–810 (1995).

[11] M. Ancona, G. Dodero, and V. Gianuzzi, ‘Efficient construction of LR(k) states and tables’, *ACM Transactions on Programming Languages and Systems*, **13**(1), 150–178 (1991).

[12] G. Merrill, ‘Parsing non-LR(k) grammars with Yacc’, *Software–Practice and Experience*, **23**(8), 829–850 (1993).

[13] J. Lee, *The anatomy of a compiler*, Van Nostrand, 1967.

[14] C. Rutzinger, 'Parsing Fortran'. section 3.4.1 of Fortran FAQ: `http://www.fortran.com/fortran/FAQ/tech.html#3.4.1`.

[15] ctags utility, 'Unix manual pages'.

[16] G. Murphy and D. Notkin, 'Lightweight source model extraction', *SIGSOFT*, 1995.

[17] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, 1976.

Appendix 1

Table I describes the articulation table for program level articulation in the Pascal parser. The top of stack is shown in the left-hand column. If more than one item on the feature stack is examined to make a decision, the tokens are shown (for example, `begin - begin` which indicates a nested begin construction). The second column is the current feature token. Only valid tokens are shown for each particular state of the stack. In the table, token `procedure` is used to represent both `procedure` and `function` as they have similar behaviour. Similarly, `label` is used to represent `label`, `const`, `type` and `var`. The action when a `record` token is seen only applies when `type` and `var` are top of the stack. It is not valid for the other two declarations.

Starting a construct pushes a new place-holder token on the construct and feature stacks. It also attaches the place-holder token to the tree, and tokens are now attached as children of the place holder until it is closed.

Appendix 2

The articulation table for statement-level articulation in the Pascal parser is shown in Table II.

The actions are slightly more complicated than those of the program-level articulation table due to nested statements and the need to check that terminators for statements such as `end`, `else` and `until` appear correctly nested.

top of stack	valid next feature token	action
start of file	program	start program declaration; push program
program	label	start label declaration; push label
	procedure	start undetermined declaration; push procedure
	begin	start begin-end; push begin
	(or)	no action
label	label	end label declaration; start label declaration; push label
	procedure	end label declaration; start undetermined declaration; push procedure
	begin	end label declaration; start begin-end; push begin
	record	push record;
	(or)	no action
procedure	(	start parentheses; push (
	label	start label declaration; push label
	procedure	start undetermined declaration; push procedure
	forward	change construct to procedure decl; push forward
	begin	change construct to procedure defn; start begin-end; push begin
forward	label	end procedure decl; start label declaration; push label
	procedure	end procedure decl; start undetermined declaration; push procedure
	begin	end procedure decl; start begin-end; push begin
begin-end - begin	begin	push begin
	case	push case
	end	end begin-end; push begin; push end
	(or)	no action
begin - begin case - begin	begin	push begin
	case	push case
	end	pop begin
	(or)	no action
program - begin - end	end of file	end program declaration; stop articulation
procedure - begin - end	label	end procedure decl; start label declaration; push label
	procedure	end procedure decl; start undetermined declaration; push procedure
	begin	end procedure decl; start begin-end; push begin
(	(	start parentheses; push (
	var or procedure	no action
	)	end parentheses
case	begin	push begin
	case	push case
	end	pop case
	(or)	no action
record	record	push record
	end	pop record
	case, (or)	no action

Table I. Articulation table for program level.

top of stack	valid next feature token	action
undetermined statement	:=	change construct to assign stmt; push :=
	begin	change construct to compound stmt; push begin push undetermined statement
	case	change construct to case stmt; push case
	:	change construct to labelled stmt; push : start undetermined statement
	else	check within if-then construct; identify statement; close current statement up to then; change construct to if-then-else stmt; push else; start undetermined statement
	end	identify statement; close current statement up to begin or case; push end
	if	change construct to if-then stmt; push if
	goto	change construct to goto stmt; push goto
	(	start parentheses; push (
	repeat	change construct to repeat stmt; push repeat start undetermined statement
	;	identify statement; close current statement start undetermined statement
	until	check within repeat construct; identify statement; close current statement up to repeat; push until
	while	change construct to while stmt
	with	change construct to with stmt
assign-stmt - := begin - end case - of - end goto else until	else	check within if-then construct; close current statement up to then; change construct to if-then-else stmt; push else; start undetermined statement
	end	check within compound construct; close current statement up to begin or case; push end
	;	close current statement start undetermined statement
	until	check within repeat construct; close current statement up to repeat; push until
case	of	push of
case - of	:	push :; start undetermined statement
case - of	end	push end
for	:=	push :=
for - :=	to	push to
for - := - to	do	push do; start undetermined statement
if	then	push then; start undetermined statement
while	do	push do; start undetermined statement
with	do	push do; start undetermined statement

Table II. Articulation table for statement level.