Recursive-Descent Parsing

Ian J. Hayes

March 4, 2012

An Extended BNF grammar for a language can be used to devise a parser for the language. Section 1 describes notations for presenting the formal definition of the syntax of a language: Backus Naur Form (BNF) and Extended Backus Naur Form (EBNF). A grammar in EBNF form is particularly suitable to use as a basis for writing a recursive-descent parser for a language (Section 2). Section 4 discusses how to handle syntax error recovery and Section 5 discusses how to build an Abstract Syntax Tree (AST) while parsing. In Section 6 we define the basic theory of grammar derivations, including the set of tokens that can start a grammar construct as well as the set of tokens that can follow a construct in a particular context; these concepts are needed in order to describe whether or not a grammar is suitable for predictive recursive-descent parsing.

1 Extended Backus Naur Form

Backus Naur Form (BNF) was invented by John Backus as an elegant notation for describing the syntax of a programming language and was first used to describe the syntax of Algol [1].[1] The notation was slightly modified by Peter Naur and popularised via the report on the programming language Algol 60 edited by Naur [5].

BNF allows a *context-free grammar*[2] to be described by a set of productions, each of the form $N \to \alpha$, where the left side consists of a single nonterminal, N, and the right side consists of a possibly empty sequence, α, of terminal and nonterminal symbols.[3] For example, a production in the syntax for an expression follows.

$$Exp \to Exp \text{ `+' } Term$$

To allow the grammar to be presented more succinctly, BNF allows the right side of a production to consist of a set of alternatives separated by '|', as follows

$$N \to \alpha \mid \beta \mid \ldots \mid \gamma$$

although this doesn't add any expressive power because it could have been written as a set of productions for N, one for each right side alternative.

[1] BNF was originally known as Backus Normal Form [3].

[2] Context-free grammars correspond to Chomsky Type 2 grammars [2] and are also known as *phrase-structured grammars*.

[3] The original BNF syntax used "::=" instead of "→", and nonterminals were placed in angle brackets to distinguish them from other characters, which stood for themselves; the production for an expression was written as follows.

$$\langle Exp \rangle \ ::= \ \langle Exp \rangle \ + \ \langle Term \rangle$$

Extended BNF (EBNF) is a notation invented by Niklaus Wirth to allow grammars to be expressed even more succinctly [7]. It extends BNF to allow an optional syntactic construct [in square brackets], a repetition of a construct {in curly braces}, and grouping of constructs (in parentheses). A grammar for Extended BNF, given in Extended BNF, is presented in Figure 1.[4] Figure 2 gives an example EBNF grammar, which defines the syntax of conditionals and expressions.

BNF equivalents of EBNF extensions. The additional notation in Extended BNF does not add any expressive power because we can always translate an EBNF grammar to a BNF grammar. We show how the EBNF optional, repetition, and grouping constructs can be eliminated from an EBNF grammar to leave a BNF grammar.

An optional construct [S] occurring in the right side of a grammar rule is replaced by a new nonterminal,[5] *OptS*, defined as follows:

$$OptS \to S \mid \epsilon \tag{1}$$

where ϵ is the empty string.[6] For example, the rule for *Condition* in Figure 2 can be rewritten as

$$\begin{array}{rcl} Condition & \to & Exp \ OptRelExp \\ OptRelExp & \to & RelOp \ Exp \mid \epsilon \end{array}$$

Aside: This example could also have been written as

$$Condition \quad \to \quad Exp \mid Exp \ RelOp \ Exp$$

but we prefer to use the more general rule for optionals because it works in any situation.

A repetition construct { S } occurring in the right side of a grammar rule is replaced by a new nonterminal, *RepS*, defined as follows:

$$RepS \to \epsilon \mid (S) \ RepS \tag{2}$$

The grouping parentheses around the S are needed in case S consists of a set of alternatives, e.g., $S \to A \mid B$, in which case leaving out the parentheses gives

$$RepS \to \epsilon \mid A \mid B \ RepS \tag{3}$$

which is wrong because concatenation has higher precedence than alternation; the correct version follows.

$$RepS \to \epsilon \mid (A \mid B) \ RepS \tag{4}$$

[4] The original EBNF used "=" instead of "→", and terminated each production with ".".

[5] Any fresh (i.e., unused) name will do.

[6] Strictly speaking, if S is an alternation, e.g., $S \to A \mid B$, then S should be in grouping parentheses giving $OptS \to (A \mid B) \mid \epsilon$ but because alternation is associative, this is equivalent to $OptS \to A \mid B \mid \epsilon$ and hence the grouping parentheses are not required.

Grammar → *Production* { *Production* }
Production → *Nonterminal* '→' *EBNFExpression*
EBNFExpression → *EBNFTerm* { '|' *EBNFTerm* }
EBNFTerm → { *EBNFFactor* }
EBNFFactor → *Terminal* | *Nonterminal* | '[' *EBNFExpression* ']' | '{' *EBNFExpression* '}' | '(' *EBNFExpression* ')'
Nonterminal → *IDENTIFIER*
Terminal → *STRING* | *IDENTIFIER*

Figure 1: Grammar for Extended BNF (in Extended BNF)

Terminal symbols are all-upper-case identifiers.

Condition → *Exp* [*RelOp Exp*]
RelOp → *EQUALS* | *NEQUALS* | *LEQUALS* | *LESS* | *GREATER* | *GEQUALS*
Exp → [*PLUS* | *MINUS*] *Term* {(*PLUS* | *MINUS*) *Term*}
Term → *Factor* {(*TIMES* | *DIVIDE*) *Factor*}
Factor → *LPAREN Condition RPAREN* | *NUMBER* | *LValue*
LValue → *IDENTIFIER*

Figure 2: An EBNF grammar for expressions

For example, the rule for *Term* in Fig. 2 can be rewritten as

Term → *Factor RepF*
RepF → ((*TIMES* | *DIVIDE*) *Factor*) *RepF* | ϵ

In this case the outer parentheses introduced by the rule are redundant and can be omitted.

A grouping construct (*S*) occurring in the right side of a grammar rule is replaced by a new nonterminal, *GrpS*, defined as follows:

$$GrpS \rightarrow S \tag{5}$$

For example, the rule for *RepF* above can be rewritten as

RepF → *TermOp Factor RepF* | ϵ
TermOp → *TIMES* | *DIVIDE*

Exercise 1. Rewrite the rule for *Exp* in Fig. 2 to remove optionals, repetitions, and grouping parentheses.

2 Recursive-descent parsing

Provided the EBNF grammar for a language is in a restricted form (see the discussion of handling alternatives below) it can be used to derive a recursive-descent parser,[7] using an approach proposed by Donald Knuth [4]. A recursive-descent parser is a recursive program to recognise sentences in the language. A recursive-descent parser consists of a set of methods, one for each nonterminal symbol. We'll give our examples in Java but a recursive-descent parser can be written in any programming language that supports recursion.

The input is assumed to be a stream of lexical tokens (also known as terminal symbols) and the parse begins with the current token being the first token in the input stream. For each nonterminal symbol, *N*, there is a method called "`parseN`", which recognises the longest string of terminal symbols in the input stream, starting from the current token, which can be derived from the nonterminal *N*. As it parses the input it moves the current position forward, and when it has finished parsing *N*, the current token is the token immediately following the last token matched as part of *N*. For the expression grammar given in Figure 2 we need a parse method for each nonterminal. Figure 3 gives these parsing methods except for `parseRelOp`, which is left as an exercise, and `parseLValue`, which is given in the text below.[8]

Recognising a nonterminal. To recognise a nonterminal, *N*, the parser simply calls the corresponding method:

```
parseN();
```

For example, to parse a *Condition* in the expression grammar we call `parseCondition()`.

Recognising a terminal. To recognise a terminal symbol, *T*, the parser calls the method `match`, with the terminal symbol *T* as a parameter:

```
match(T);
```

If the current input token is *T*, the match is successful and the position in the input stream is moved to the next token.[9] If the current token isn't *T*, a syntax error message is generated. We discuss syntax error recovery in Section 4. For example,

[7] More fully an LL(1) recursive-descent predictive parser.

[8] In this presentation we have omitted Java qualifiers like `private` and `public` to avoid cluttering the presentation.

[9] In the PL0 compiler, if the parse debugging flag is set, `match` also outputs some parse tracing information.

the parsing method for the nonterminal *LValue* in Figure 2 matches a single *IDENTIFIER* token.

```
void parseLValue() {
    match( IDENTIFIER );
}
```

Recognising a sequence. A sequence, $S_1\ S_2\ \ldots\ S_n$, of EBNF terms is recognised by code that recognises each in sequence.

$$\begin{array}{l} recog(S_1); \\ recog(S_2); \\ \ldots \\ recog(S_n); \end{array}$$

where $recog(S_i)$ is the code to recognise the EBNF term S_i (which depends on the form of the term S_i). If the sequence of EBNF terms to be matched is empty then no code is required to match it.

For example, one alternative for a *Factor* in the expression grammar is the sequence

LPAREN Condition RPAREN

which can be recognised by the following code sequence.

```
match( LPAREN );
parseCondition();
match( RPAREN );
```

Recognising alternatives. To recognise a set of EBNF alternatives, $S_1 \mid S_2 \mid \ldots \mid S_n$, we need to first determine which alternative to use. Our (predictive) recursive-descent parser predicts which alternative to recognise based on just the value of the current token in the input stream. For a particular alternative, S_i, to be chosen either

- the current token is in the set of terminal symbols, $First(S_i)$, that can start S_i, or
- S_i is nullable and the current token can validly follow in the context of the complete set of alternatives.

Section 6 discusses how to calculate the sets of symbols that can start a grammar construct and follow a nonterminal.

For this approach to work, we must restrict the language grammar so that

- any given token is in the first set of at most one alternative, i.e., the sets of first symbols of each pair of alternatives are disjoint,
- at most one alternative is nullable, and
- if there is a nullable alternative, the set of symbols that may follow in the context of the complete set of alternatives is disjoint from the first sets of all alternatives.[10]

[10] Occasionally we ignore this restriction and give precedence to matching the non-nullable alternatives.

Section 6 formalises the conditions for a grammar to be suitable for predictive recursive descent parsing.

If none of the alternatives is nullable, the code to parse the set of alternatives is as follows, in which the variable `token` contains the current token.

```
if( token.isIn(First(S1)) ) {
    recog(S1);
} else if( token.isIn(First(S2)) ) {
    recog(S2);
} else ...
    ⋮
} else if( token.isIn(First(Sn)) ) {
    recog(Sn);
} else {
    error("Syntax error");
}
```

For example, the parsing method for a *Factor* given in Figure 3 is of this form; in this case the first set of each alternative consists of just one token, and hence we have used `isMatch` instead of `isIn`.

If one of the alternatives can match empty, the code for the recogniser is the same as the above, except that we leave off the last "else" clause (which gives an error message).[11]

Exercise 2. Write the code to recognise the nonterminal *RelOp* in the grammar given in Figure 2.

Recognising an optional. An EBNF optional construct is of the form, $[\ S\ ]$, where S is an EBNF expression. It can either match S or the empty sequence, and hence it is similar to matching alternatives. The code to recognise $[\ S\ ]$ follows.

```
if( token.isIn(First(S)) ) {
    recog(S);
}
```

To be unambiguous the first set of S should be disjoint from the set of tokens that can follow the optional construct.

For example, the production for *Condition* contains an optional. The code for the parser for a *Condition* is given in Figure 3. As another example, consider the grammar for an if-then-else statement in which the else-part is optional.

$$\begin{array}{rcl} \mathit{IfS} & \rightarrow & \texttt{`if'}\ C\ \texttt{`then'}\ S\ [\ \texttt{`else'}\ S\ ] \\ S & \rightarrow & \mathit{IfS} \mid \ldots \end{array}$$

The nonterminal C stands for a condition and the nonterminal S stands for a statement, where a statement may be an if-statement or some other form of statement. An equivalent BNF grammar for an if-statement is as follows.

$$\begin{array}{rcl} \mathit{IfS} & \rightarrow & \texttt{`if'}\ C\ \texttt{`then'}\ S\ \mathit{ElsePart} \\ \mathit{ElsePart} & \rightarrow & \texttt{`else'}\ S \mid \epsilon \end{array}$$

This grammar is ambiguous because for a string like

[11] Instead of this, one could check whether the current token is in the follow set for the construct and if it isn't generate an error message. In practice just doing nothing is fine because an error will be detected when we try to recognise the following constructs. Our syntax error recovery process described in Section 4 also handles this case.

REL_OPS_SET consists of the set of terminal symbols representing relational operators, EXP_OPS_SET is the set containing PLUS and MINUS, and TERM_OPS_SET is the set containing TIMES and DIVIDE.

```
void parseCondition() {
    parseExp();
    if( token.isIn( REL_OPS_SET ) ) {
        parseRelOp();
        parseExp();
    }
}
void parseExp() {
    if( token.isMatch(PLUS) ) {
        match( PLUS );
    } else if( token.isMatch(MINUS) ) {
        match( MINUS );
    }
    parseTerm();
    while( token.isIn( EXP_OPS_SET ) ) {
        if( token.isMatch(PLUS) ) {
            match( PLUS );
        } else if( token.isMatch(MINUS) ) {
            match( MINUS );
        } else {
            fatal( "unreachable because of guard in while" );
        }
        parseTerm();
    }
}
void parseTerm() {
    parseFactor();
    while( token.isIN( TERM_OPS_SET ) ) {
        if( token.isMatch(TIMES) ) {
            match( TIMES );
        } else if( token.isMatch(DIVIDE) ) {
            match( DIVIDE );
        } else {
            fatal( "unreachable because of guard in while" );
        }
        parseFactor();
    }
}
void parseFactor() {
    if( token.isMatch( LPAREN ) ) {
        match( LPAREN );
        parseCondition();
        match( RPAREN );
    } else if( token.isMatch( NUMBER ) ) {
        match( NUMBER );
    } else if( token.isMatch( IDENTIFIER ) ) {
        parseLValue();
    } else {
        error( "Syntax error" );
    }
}
```

Figure 3: Parsing methods for a conditions

$$
\begin{array}{rcl}
\mathit{Exp} & \rightarrow & [\mathit{PLUS} \mid \mathit{MINUS}]\ \mathit{Term}\ \{(\mathit{PLUS} \mid \mathit{MINUS})\ \mathit{Term}\} \\
\mathit{Term} & \rightarrow & \mathit{Factor}\ \{(\mathit{TIMES} \mid \mathit{DIVIDE})\ \mathit{Factor}\} \\
\mathit{Factor} & \rightarrow & \mathit{LPAREN}\ \mathit{Exp}\ \mathit{RPAREN} \mid \mathit{NUMBER}
\end{array}
$$

Figure 4: Simple grammar for calculator expressions

```
if c0 then if c1 then s0 else s1
```

the else-part can be linked to either the first '`if`' or the second '`if`', as shown by the grouping parentheses in the following (where the parentheses are not part of the syntax).

```
if c0 then (if c1 then s0) else s1
if c0 then (if c1 then s0 else s1)
```

This is known as the "dangling-else" problem. The grammar breaks the rule given above for alternatives, which requires that if *ElsePart* is nullable, the first set for *ElsePart* must be disjoint from the follow set for *ElsePart*, but in this case both sets contain '`else`'. In practice language designers using an if-then-else of this form[12] have decided to resolve the ambiguity in the grammar by always matching an '`else`' with the closest preceding '`if`'. This gives precedence to the '`else`' alternative over the empty alternative, i.e., when recognising an *ElsePart*, the empty alternative is only used if the current token is not '`else`'. The code to match an if-then-else statement is the same as we would normally generate for an optional.

```
match("if");
parseC();
match("then");
parseS();
if( token.isMatch("else") ) {
    match("else");
    parseS();
}
```

Recognising a repetition. An EBNF repetition construct is of the form, $\{\ S\ \}$, where S is an EBNF expression. It can recognise a sequence of zero or more occurrences of S. The code to recognise $\{\ S\ \}$ consists of a loop.

```
while( token.isIn(First(S)) ) {
    recog(S);
}
```

To be unambiguous the first set of S should be disjoint from the set of tokens that can follow the repetition construct. The parsing methods for *Exp* and *Term* in Figure 3 contain examples of recognising a repetition.

Recognising grouping. In EBNF, parentheses are used to represent grouping and thus override the default precedence of EBNF, e.g., in EBNF expressions like $S_1\ (S_2 \mid S_3)$. In generating the recognition code for such EBNF expressions we must respect the grouping. Normally we don't have to do anything extra in the code, because the generated code already has Java grouping curly braces. The parsing method *Term* in Figure 3 contain an example of recognising a group.

Removing redundant code. Using the rules defined above the code for recognising [*PLUS* | *MINUS*] at the beginning of `parseExp` should be

```
if( token.isIn(EXP_OPS_SET) ) {
    if(token.isMatch(PLUS)){
        match(PLUS);
    } else if(token.isMatch(MINUS)){
        match(MINUS);
    } else {
        fatal( "unreachable branch" );
    }
}
```

however, it can be optimised by removing redundant checks to give the code given at the start of `parseExp` in Figure 3.

3 A simple expression calculator

The methods given so far are only recognisers. In Figure 4 we give a grammar for integer expressions, which is simplified from that given earlier, in that it doesn't include relational operators or identifiers.

Figure 5 extends the recognisers for expressions, terms and factors to calculate the value of the expression. The main difference is that the methods now all return an integer which is the value of the corresponding expression, term, or factor, and additional code is added to perform the calculations.

Exercise 3. Write the code to recognise the nonterminal *Exp* in the grammar given in Figure 4 and evaluate the recognised expression.

4 Syntax error recovery

Syntax error recovery during recursive-descent parsing can be accomplished by

- local error recovery on matching a single token; and
- synchronising the input stream at the start and end of each parse method.[13]

[12] This includes languages like Algol, Pascal, C, Java, and many others.

[13] This uses an approach based on that used by Welsh and McKeag [6].

TERM_OPS_SET is the set containing TIMES and DIVIDE.

```
int parseTerm() {
    int result;
    boolean times;
    result = parseFactor();
    while( token.isIN( TERM_OPS_SET ) ) {
        if( token.isMatch(TIMES) ) {
            match( TIMES );
            times = true;
        } else if( token.isMatch(DIVIDE) ) {
            match( DIVIDE );
            times = false;
        } else {
            fatal( "unreachable because of guard in while" );
        }
        int factor = parseFactor();
        if( times ) {
            result = result * factor;
        } else {
            result = result / factor;
        }
    }
    return result;
}
int parseFactor() {
    int result;
    if( token.isMatch( LPAREN ) ) {
        match( LPAREN );
        result = parseCondition();
        match( RPAREN );
    } else if( token.isMatch( NUMBER ) ) {
        result = token.getIntValue(); // Requires token to be a NUMBER
        match( NUMBER );    // must match after extracting the number
    } else {
        error( "Syntax error" );    // could raise an exception here
        result = 0x80808080; // something pretty useless
    }
    return result;
}
```

Figure 5: Expression calculator

Local error recovery. On matching a single token we can handle:

- a (single) token *missing* from the input stream,
- an additional token erroneously *inserted* into the input stream, or
- a single erroneous token *replacing* the expected token in the input stream.

To handle local error recovery while matching a token, T, the match method requires a second parameter, FS, that is the set of tokens that can immediately follow T in the context in which T is being matched. FS is only used when the current token doesn't match T, in which case an error message is generated and FS is used for syntax error recovery as follows:

- if the current token is in FS, the syntax error recovery is to assume that T was *missing* from the input, and hence no further action is taken;
- if the current token is not in FS, the current input token is skipped (and becomes the previous token) and then
  - if the current token is now T, the recovery is to assume that the previous token was erroneously *inserted* into the input stream, and the recovery action is to match the (new) current token, or
  - if the current token isn't T, the recovery is to assume that the previous token in the input stream was supposed to be T but it was *replaced* by an erroneous token; no further recovery action is required.

```
boolean beginRule( String rule, TokenSet expected, TokenSet recoverSet )
boolean beginRule( String rule, Token expected, TokenSet recoverSet)
void beginRule( String rule, TokenSet expected )
void beginRule( String rule, Token expected )
void endRule( String rule, TokenSet recoverSet )
```

Figure 6: Headers for methods `beginRule` and `endRule`

For example, consider matching the following production.

WhileStatement →
KW_WHILE Condition KW_DO Statement

In matching of the tokens `KW_WHILE` and `KW_DO` the second parameter to `match` consists of the set of symbols that can start the following constructs, *Condition* and *Statement*, respectively. Recovery sets are explained below.

```
void parseWhileStatement() {
  match( KW_WHILE, CONDITION_START_SET );
  parseCondition(recoverSet.union(KW_DO));
  match( KW_DO, STMT_START_SET );
  parseStatement( recoverSet );
}
```

There is still a single-parameter version of `match` available, however, it gives a fatal error if the token does not match, and hence is only suitable for use in cases where it is known that the match will definitely succeed, in which case it sets the current token to the next input token. For example, the method for matching a "while" statement is only ever called if the current token is the keyword `KW_WHILE`; in this case the first line in the above code can be replaced by the following.

```
match( KW_WHILE );     // cannot fail
```

Synchronisation at the beginning of parse methods. At the start of a parsing method for a nonterminal, N, the current token is valid if

- it can start N, i.e., it is in $First(N)$, or
- N is nullable and the current token may follow N in the context in which N is being recognised.

Before trying to match a nonterminal N, we "synchronise" the input to ensure the current token is valid using the method `beginRule`, whose header is the first for `beginRule` given in Figure 6. We use a call on `beginRule` at the start of a parsing method for N (see Figure 7). The first parameter to `beginRule` is a string containing the name of the parsing rule (i.e., nonterminal) being recognised; it is used in error (and debugging) messages. The second parameter, `expected`, is the set of tokens that are valid at the start of parsing N; unless N is nullable, `expected` is $First(N)$. The third parameter, `recoverSet`, is the set of tokens that are expected to follow N in the context in which N is being parsed. Because `recoverSet` depends on the context in which N is to be recognised, `recoverSet` is added as a parameter to the parse method for each nonterminal.

Figure 6 contains four headers for `beginRule`: the first is as described above, the second is used for the special case when `expected` is a single token, the third handles the case when the token must match (a failure to match indicates an error in the parser, not the input being parsed), and the fourth is the special case of the third when `expected` is a single token.

The parse method for N, with synchronisation added, has the form given in Figure 7.

If on a call to `beginRule`, the current token is in the set `expected`, there is no syntax error and the synchronisation method `beginRule` does nothing and returns true[14] and then the parse method will attempt to recognise N. If the current token is not in `expected` a syntax error message is generated and the recovery action is to skip tokens until either

- a token is found that is valid at the start of N (i.e., it is in `expected`), and then proceed to recognise N from that point, or else
- a token that is in `recoverSet` is found, and return as though we had recognised N.

The method `beginRule` implements this strategy by skipping tokens until a token is found that is either in `expected` or `recoverSet`. If it is in `expected` (i.e., the first case above when synchronisation succeeds), `beginRule` returns true and the parsing method for N proceeds to (try to) parse N. If after skipping tokens, the current token is not in `expected`, and hence it must be in `recoverSet` (i.e., the second case above when synchronisation fails), `beginRule` returns false and the parsing method for N exits as though it has already parsed N.

Synchronisation at the end of parse methods. At the end of the parse method for a nonterminal, N, after N has been recognised, we synchronise the input stream so that the current token is one that is expected to follow N, i.e., it is in the `recoverSet` passed to `parseN`. This is done by a call to `endRule` as shown in Figure 7.

If the current token is in the `recoverSet`, `endRule` does nothing, otherwise an error message is generated and the syntax error recovery strategy is to skip tokens until one is found that is in `recoverSet`, at which point `endRule` returns.[15]

[14] In the PL0 compiler it can also print out some debugging information if the parse debug flag ("-d") is set.

[15] The PL0 compiler also outputs debugging information if the parse debug flag is set.

```
void parseN( TokenSet recoverSet ) {
    if( !beginRule( "N", N_START_SET, recoverSet ) ) {
        return;
    }
    recog(N);
    endRule( "N", recoverSet );
}
```

Figure 7: Parse method for recognising *N* with synchronisation

The set `CONDITION_START_SET` contains the tokens `PLUS`, `MINUS`, `LPAREN`, `NUMBER`, and `IDENTIFIER`.

```
void parseCondition( TokenSet recoverSet ) {
    if( !beginRule( "Condition", CONDITION_START_SET, recoverSet ) ) {
        return;
    }
    parseExp( recoverSet.union( REL_OPS_SET ) );
    if( token.isIn( REL_OPS_SET ) ) {
        parseRelOP( recoverSet.union( EXP_START_SET ) );
        parseExp( recoverSet );
    }
    endRule( "Condition", recoverSet );
}
```

Figure 8: Parse method for recognising *Condition* with synchronisation

Accumulation of tokens in recovery sets. The recursive-descent parsing process begins by calling the parse method for the start symbol, *S*, for the language with a recovery set containing just the token representing end-of-file. That method may, for example, call the parse method for the nonterminal *A*, which may in turn call the parse method for the nonterminal *B*, which may recursively call the parse method for *A*, etc. When the parse method for *S* calls `parseA` it passes a recovery set containing the tokens that can immediately follow that *A* within *S* as well as the recovery set passed into *S*. Similarly, when `parseA` calls `parseB` it passes a recovery set containing the tokens that can immediately follow that *B* within *A* as well as the recovery set passed into *A*, and so on.

As the parser nests down in recursive calls, the recovery set always contains the recovery set for the level above and hence recovery sets only grow as the parser nests down through recursive calls. When a recovery set is created for a call, a new set object is always created, rather than modifying the existing recovery set, so that on return from a nested call, the recovery set is still the original one passed in at that level.

Because the recovery sets only grow as the parser nests down through calls, all the recovery sets contain the recovery set originally passed in to the parse method for the start symbol, which contained just the end-of-file token, and hence the token skipping for syntax error recovery in both `beginRule` and `endRule` can't skip past end-of-file.

Fig. 8 gives the parse method for the *Condition* in the grammar given in Fig. 2, complete with error recovery. The set `CONDITION_START_SET` contains the tokens that may start a *Condition* (see Fig. 8). This set is passed to `beginRule` as the set of tokens that are expected at the start of a *Condition*.

The recovery set for the first call to `parseExp` consists of the recovery set passed into `parseCondition` unioned with the set of all relational operators, because a relational operator may immediately follow that occurrence of *Exp* in the production for *Condition*. Similarly, the recovery set for `parseRelOp` is `recoverSet` unioned with the tokens that may start an *Exp* (which happen to be the same as the tokens that can start a *Condition*). The recovery set for the second call to `parseExp` only contains the recovery set passed into `parseCondition` because that occurrence of *Exp* in the production for *Condition* occurs at the end of the production. Note that the two calls to `parseExp` in Figure 8 use different recovery sets because the set of tokens that can follow in the context of the first occurrence of *Exp* differs from the set of tokens which can follow the second occurrence of *Exp*.

Note that the *recovery set* for a parse method is not the same as the *follow set* for the nonterminal it is recognising. The follow set for a nonterminal, *N*, can be calculated statically from the grammar and contains all tokens that can follow *N* in any context. By comparison, the recovery set for a call to `parseN` contains the tokens that can follow *N* in the context of that occurrence of *N* (not all contexts) — this part of the recovery set is contained in the follow set of *N* — but the recovery set also contains the recovery sets for all the higher-level calls on parse methods that are still active, all the way up to the initial call on the parse method for the start symbol (for which the recovery set just contained the end-of-file token).

TERM_OPS_SET is the set containing TIMES and DIVIDE, and TERM_START_SET is the set of tokens that can start a *Term*, i.e., LPAREN, NUMBER, and IDENTIFIER.

```
ExpNode parseTerm( TokenSet recoverSet ) {
    if( !beginRule("Term", TERM_START_SET, recoverSet) ) {
        return new ExpNode.ErrorNode( token.getPosn() );
    }
    ExpNode term = parseFactor( recoverSet.union(TERM_OPS_SET) );
    while( token.isIN( TERM_OPS_SET ) {
        BinaryOperator operator = BinaryOperator.INVALID_OP;
        Position opPosition = token.getPosn();
        if( token.isMatch(TIMES) ) {
            operator = BinaryOperator.MUL_OP;
            match( TIMES );
        } else if( token.isMatch(DIVIDE) ) {
            operator = BinaryOperator.DIV_OP;
            match( DIVIDE );
        } else {
            fatal( "Unreachable branch in parseTerm" );
        }
        ExpNode right = parseFactor( recoverySet.union(TERM_OPS_SET) );
        term = new ExpNode.BinaryOpNode( opPosition, operator, term, right );
    }
    endRule( "Term", recoverSet );
    return term;
}
```

Figure 9: Parsing method for *Term* with AST building

5 Building an AST

As well as parsing a language, we would like to create a representation of the language in the form of an abstract syntax tree (AST). To do this we change the parse method for a nonterminal, *N*, so that it returns the abstract syntax tree representation of *N*, assuming that all the parse methods it calls return the abstract syntax tree representations of the nonterminals they recognise. For example, the parse method for a *Term* in the grammar given in Figure 2 is given in Figure 9.

$$\mathit{Term} \rightarrow \mathit{Factor}\ \{(\mathit{TIMES} \mid \mathit{DIVIDE})\ \mathit{Factor}\ \}$$

We have also included syntax error recovery so that this is the final version of this parse method.

The method returns an ExpNode as described in the PL0 Compiler Data Structures document, and as used in the PL0 compiler. If the call to beginRule fails to synchronise (returns false) then we still need to return an ExpNode, so we return an ErrorNode. The error node has the current position in the source file associated with it.

We assume that the method parseFactor also returns an ExpNode. The result of the first call on parseFactor is saved in the local variable term, in which we progressively build up the tree representing the term as it is recognised. If there is no TIMES or DIVIDE token following the factor, the while loop is not entered, and the value of term is returned.

If there is a TIMES or DIVIDE, the while loop is entered and the operator is matched. We also remember which operator was matched and its position in the source input. The conditional statement that matches TIMES or DIVIDE has a redundant error alternative which cannot be reached because of the guard in the repetition. This redundant code calls method fatal, which flags a fatal error in the compiler itself, not the program it is compiling. This redundant code is left in the compiler in order to pick up errors in the compiler as quickly as possible. A common situation in which this occurs is when compiler code is cut-and-pasted to form a new parsing method, but the pasted method isn't updated consistently for the new construct. The PL0 compiler contains many occurrences of such redundant code left in the compiler to pick up compiler errors.

After matching the operator, we parse its right hand factor, placing the result in the variable right. Note that the syntax error recovery set for this call includes TERM_OPS_SET because there could be another TIMES or DIVIDE following the factor, which will be recognised by the next iteration of the loop. Once we have recognised the second factor, we build a BinaryOpNode from the position of the operator, the operator, the first operand term, and the second operand right. The new BinaryOpNode is assigned to term. If the loop exits at this point, it returns this value of term, but if the loop repeats another operator and factor will be recognised and a new BinaryOpNode will be constructed using the value of term as its left operand and the new factor as its right operand. This has the effect of treating TIMES and DIVIDE as left associative operators with the same precedence.

Exercise 4. Give code to recognise both *Exp* and *Factor* in the grammar given in Figure 2 and build abstract syntax trees (*ExpNode*) representing them; your code should include syntax error recovery.

6 First, follows and LL(1) grammars

Productions can be applied to rewrite strings of terminal and nonterminal symbols by replacing a nonterminal symbol with the right side of one of its productions.

Definition 1 (Direct derivation step) *Let both α and β be possibly empty strings of terminal and nonterminal symbols, and N a nonterminal symbol. If there is a production of the form $N \rightarrow \gamma$, then a* derivation step *can be applied to a string of symbols of the form $\alpha\ N\ \beta$ to replace the N by the string γ to give the string $\alpha\ \gamma\ \beta$. We say $\alpha\ N\ \beta$* directly derives *$\alpha\ \gamma\ \beta$, written*

$$\alpha\, N\, \beta \;\Rightarrow\; \alpha\, \gamma\, \beta\,.$$

Note the use of the double arrow "⇒" for a derivation versus a single arrow "→" within a production.

Definition 2 (derives) *We say a string of terminal and nonterminal symbols α* derives *a string β, written*

$$\alpha \stackrel{*}{\Rightarrow} \beta$$

if there is a sequence of zero of more direct derivation steps starting from α and finishing with β. That is there must exist a sequence of one or more strings, $\gamma_0, \gamma_1, \ldots, \gamma_n$, such that, $\alpha = \gamma_0$, $\gamma_n = \beta$, and for each i between 1 and n, $\gamma_{i-1} \Rightarrow \gamma_i$, i.e., $\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \beta$.

Definition 3 (Nullable) *A syntactic construct, α, is nullable if it can derive the empty string of symbols, i.e. $\alpha \stackrel{*}{\Rightarrow} \epsilon$.*

Obviously ϵ is nullable and any terminal symbol is not nullable. A sequence of the form $S_1\ S_2\ \ldots\ S_n$ is nullable provided all of the constructs $S_1, \ldots, S_n$ are nullable. A set of alternatives $S_1 \mid \ldots \mid S_n$ is nullable if at least one of the alternatives is nullable. EBNF constructs for optionals and repetitions are nullable. A nonterminal *N* is nullable if there is a production for *N* with a nullable right side.

In predictive recursive-descent parsing we make choices between alternatives based on the current token. To do this we need to know which tokens each alternative can begin with; this set of tokens is called its *First* set. If a construct is nullable, in order to choose between recognising the empty string or a nonempty string, we also need to know which symbols can *follow* the construct.

First sets. In addition to containing the set of tokens the construct can begin with, the first set also records whether or not the construct is nullable by including the empty string, ϵ, in its first set if and only if the construct is nullable. It should be emphasised that the ϵ is not recording that the construct can begin with the empty string, but rather the whole construct can match the empty string, i.e., it is nullable.

Note that ϵ is not a terminal symbol but the empty string of terminal symbols, and hence not all elements of a first set are necessarily terminal symbols. Including ϵ in the first set is a bit confusing. It is there purely to indicate that the construct is nullable, and hence the first set encodes two bits of information: the terminal symbols that can start the construct and whether or not it is nullable.[16] For these definitions α, β, and γ are assumed to be possibly empty sequences of terminal and nonterminal symbols.

Definition 4 (First set) *If α is not nullable, its first set is the set of terminal symbols that can start α, i.e., those terminal symbols 'a' such that α can derive a string beginning with 'a'.*

$$\mathit{First}(\alpha) = \{a : \mathit{Terminal} \mid (\exists\, \beta \bullet \alpha \stackrel{*}{\Rightarrow} a\ \beta)\}$$

If α is nullable, First(α) includes ϵ:

$$\mathit{First}(\alpha) = \{a : \mathit{Terminal} \mid (\exists\, \beta \bullet \alpha \stackrel{*}{\Rightarrow} a\ \beta)\} \cup \{\epsilon\}$$

Calculating first sets. The first set of a terminal symbol '*a*' is the singleton set {'*a*'}. For a sequence of constructs $S_1\ S_2\ \ldots\ S_n$, the first set obviously contains the first set of S_1, but if S_1 is nullable it also contains the first set of S_2, and if both S_1 and S_2 are nullable it also contains the first set of S_3, and so on. If $S_1\ S_2\ \ldots\ S_n$ is nullable, i.e., each of $S_1, \ldots, S_n$ is nullable, then its first set also contains ϵ.

For a set of alternatives $S_1 \mid S_2 \mid \ldots \mid S_n$, the first set contains the first sets of every alternative.

$$\begin{aligned}&\mathit{First}(S_1 \mid S_2 \mid \ldots \mid S_n) = \\ &\quad \mathit{First}(S_1) \cup \mathit{First}(S_2) \cup \ldots \cup \mathit{First}(S_n)\end{aligned}$$

If any of the alternatives is nullable, its first set will contain ϵ, and hence the first set of the set of alternatives will contain ϵ.

The first sets of optionals, repetitions, and groups are straightforward:

$$\begin{aligned}\mathit{First}([\ S\ ]) &= \mathit{First}(S) \cup \{\epsilon\} \\ \mathit{First}(\{\ S\ \}) &= \mathit{First}(S) \cup \{\epsilon\} \\ \mathit{First}((\ S\ )) &= \mathit{First}(S)\end{aligned}$$

To calculate the first set for a syntactic construct, we need to know the first sets for the nonterminals occurring in it. Hence we start by showing how to calculate the first sets of all the nonterminals in a grammar. We start with the first sets for all nonterminals set to empty and the first set for every terminal symbol '*a*' being the singleton set $\{a\}$.

We then make a pass over all productions in a grammar considering all alternatives and process as follows. If there is a production of the form $N \rightarrow \epsilon$, we add ϵ to the first set for *N* to indicate it is nullable. If there is a production of the form $N \rightarrow S_1\ S_2 \ldots S_n$, then for each $i \in 1..n$, if for all $j \in 1..i-1$,

[16] It would be better to just have the first sets containing just terminal symbols, and separately determine whether a construct is nullable, but the convention of including ϵ in the first set is almost universal within the literature and text books, and hence we won't go against that convention here.

S_j is nullable, we add the current first set for S_i to the first set for N. If every construct $S_1, \ldots, S_n$ is nullable, we add ϵ to the first set for N.

After making a complete pass in which we process every alternative right side for all productions, we repeat the process of making a pass but start with the first sets computed so far rather than the initial first sets. This pass may or may not extend some first sets. If no first sets are modified in the pass, we are finished, otherwise we repeat this process.

Because all the first sets are finite, and each time we decide to repeat the process at least one first set must have been extended by at least one symbol, the whole process must terminate. The above iterative algorithm is a common approach to calculating inductively defined sets.

Follow sets. The *Follow* set for a nonterminal, N, is the set of terminals symbols that may follow N at any context within the grammar. This requires that we look at all the contexts in which N is used, beginning from the initial context of the start symbol, S, for the grammar. One token that can follow the start symbol S is the end-of-file token, which for the purposes of this section we represent as a '`$`'.

Definition 5 (Follow) *Any nonterminal, N, can be followed by a terminal symbol 'a', if there is a derivation from S$, (i.e., the start symbol followed by the end-of-file terminal symbol), in which the terminal symbol 'a' follows N.*

$$\mathit{Follow}(N) = \{a : \mathit{Terminal} \mid (\exists\, \alpha, \beta \bullet S\$ \stackrel{*}{\Rightarrow} \alpha N a \beta)\}$$

Note that follow sets only include terminal symbols, and may include the special terminal symbol end-of-file, which only appears at the end of an input string. Note that, unlike first sets, follow sets never include ϵ.

Calculating follow sets. To handle end-of-file ('`$`') correctly, if the start symbol for the grammar is S, we add a production of the form

$$S' \rightarrow S\ \$$$

where S' is a fresh nonterminal symbol, which becomes the new start symbol for the updated grammar. We can compute the follow set for a nonterminal, N, using two facts.

1. If there is a production of the form

   $$A \rightarrow \alpha\ N\ \beta$$

   then any symbols that can start β can follow N, and hence $\mathit{Follow}(N)$ must include all the terminal symbols in $\mathit{First}(\beta)$, but note that if ϵ is in $\mathit{First}(\beta)$, it is not included in the follow set (because it is not a terminal symbol):

   $$\mathit{First}(\beta) - \{\epsilon\} \subseteq \mathit{Follow}(N)\ .$$

2. If β is nullable then any token that can follow A can also follow N, and hence

   $$\mathit{Follow}(A) \subseteq \mathit{Follow}(N)\ .$$

   The case when β is nullable includes the case when β is empty and the production is of the following form.

   $$A \rightarrow \alpha\ N$$

These facts can be used to compute the follow sets of all the nonterminals of a grammar. The process used is to start with empty follow sets for all nonterminals, except that the start symbol S has a follow set of $\{\$\}$. We make a pass through the grammar examining every alternative right side of every production. For each occurrence of a nonterminal within the right side of some production, we augment the follow set for that nonterminal according to the following process. Assuming we are processing an occurrence of N and the production is of the form

$$A \rightarrow \alpha\ N\ \beta$$

we add $\mathit{First}(\beta) - \{\epsilon\}$ to the follow set computed for N so far, and if β is nullable, we also add the current follow set for A to the current follow set for N. Caution: a common confusion is to swap the roles of A and N when applying this rule.

After making a complete pass in which we process every occurrence of a nonterminal on the right side of a production, we repeat the process of making a pass but start with the follow sets computed so far rather than the initial (mostly empty) follow sets. This pass may or may not extend some follow sets. If no follow sets are modified in the pass, we are finished, otherwise we repeat this process.

Because all the follow sets are finite, and each time we decide to repeat the process at least one follow set must have been extended by at least one symbol, the whole process must terminate.

LL(1) Grammars. Above we presented some restrictions on grammars to ensure they are suitable for recursive-descent predictive parsing. The class of grammars that are suitable is referred to as LL(1), where the first "L" refers to the fact that the parsing of the input is from **L**eft to right, the second "L" refers to the fact that they produce a **L**eftmost derivation sequence, and the "1" indicates that their parsers use one symbol lookahead (i.e., `token` is the single token lookahead).

Definition 6 (LL(1) Grammar) *A BNF grammar is LL(1) if for each nonterminal, N, the first sets for each pair of alternative productions for N are disjoint, and if N is nullable, $\mathit{First}(N)$ and $\mathit{Follow}(N)$ are disjoint.*

Because the first set for an alternative includes ϵ if the alternative is nullable, the constraint that the first sets of all the alternatives are pairwise disjoint implies that at most one alternative is nullable. Given these constraints, during recursive descent parsing the current token (i.e., the lookahead symbol) is either:

- in the first set of just one alternative and that alternative is chosen,
- if N is nullable and the current token is in the follow set of N, the nullable alternative for N is chosen, or

- if neither of the above hold, there is a syntax error.

An EBNF grammar can be considered LL(1) if when converted to a BNF grammar using the rules described in Section 1, the resulting BNF grammar is LL(1).

References

[1] J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, page 125. UNESCO, June 1959.

[2] Noam Chomsky. On certain formal properties of grammars. *Inform. Control*, 2:137–167, 1959.

[3] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Commun. ACM*, 7(12):735–736, 1964.

[4] Donald E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971.

[5] P. Naur (Ed.). Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[6] Jim Welsh and Michael McKeag. *Structured System Programming*. Prentice Hall, 1980.

[7] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

A Solutions to exercises

Solution 1. The solution requires a number of successive applications of the rules; we just give the final result here.

$$\begin{aligned} \mathit{Exp} &\rightarrow \mathit{OptPlusMinus\ Term\ RepTerm} \\ \mathit{OptPlusMinus} &\rightarrow \mathit{PlusMinus} \mid \epsilon \\ \mathit{PlusMinus} &\rightarrow \mathit{PLUS} \mid \mathit{MINUS} \\ \mathit{RepTerm} &\rightarrow \mathit{PlusMinus\ Term\ RepTerm} \mid \epsilon \end{aligned}$$

Solution 2. The method to parse a relational operator follows.

```
void parseRelOp() {
   if( token.isMatch(EQUALS) ) {
     match( EQUALS );
   } else if( token.isMatch(NEQUALS) ) {
     match( NEQUALS );
   } else if( token.isMatch(LEQUALS) ) {
     match( LEQUALS );
   } else if( token.isMatch(LESS) ) {
     match( LESS );
   } else if( token.isMatch(GREATER) ) {
     match( GREATER );
   } else if( token.isMatch(GEQUALS) ) {
     match( GEQUALS );
   } else {
     error( "Syntax error: ..." );
   }
}
```

This may be optimised to

```
void parseRelOp() {
   if( token.isIn( REL_OPS_SET) ) {
     match( token.getKind() );
   } else {
     error( "Syntax error: ..." );
   }
}
```

where `token.getKind()` returns what kind the current token is, which must be in `REL_OPS_SET` because of the guard on the if.

Solution 3. See Figure 10.

Solution 4. See the corresponding parsing methods in `Parser.java` in the PL0 compiler.

`EXP_OPS_SET` is the set containing `PLUS` and `MINUS`.

```
int parseExp() {
    int result;
    boolean minus = false; // if no sign assume plus.
    if( token.isMatch(PLUS) ) {
        match( PLUS );
    } else if( token.isMatch(MINUS) ) {
        match( MINUS );
        minus = true;
    }
    result = parseTerm();
    if( minus ) {
        result = -result;
    }
    while( token.isIn( EXP_OPS_SET ) ) {
        if( token.isMatch(PLUS) ) {
            match( PLUS );
            minus = false;
        } else if( token.isMatch(MINUS) ) {
            match( MINUS );
            minus = true;
        } else {
            fatal( "unreachable because of guard in while" );
        }
        int term = parseTerm();
        if( minus ) {
            result = result - term;
        } else {
            result = result + term;
        }
    }
    return result;
}
```

Figure 10: Expression calculator