

COMP4403 Crib Sheet

Parsing Theory

Context Free Grammars

Basic Example of Context Free Grammar

$$E \rightarrow E \text{ Op } E$$
$$E \rightarrow "(E)"$$
$$E \rightarrow \textit{number}$$
$$\text{Op} \rightarrow "+"$$
$$\text{Op} \rightarrow "-"$$
$$\text{Op} \rightarrow "*"$$

Has start symbol E , nonterminals $\{E, \text{Op}\}$, and terminals

$$\{ \backslash (" , \backslash) ", \textit{number}, \backslash + ", \backslash - ", \backslash * " \}$$

A context-free grammar consists of:

- A finite set, Σ , of terminal symbols.
- A finite nonempty set of nonterminal symbols (disjoint from the terminal symbols).
- A finite nonempty set of productions of the form of $A \rightarrow \alpha$, where A is a nonterminal symbol, and α is a possibly empty sequence of symbols, each of which is either a terminal or nonterminal symbol.
- A start symbol that must be a nonterminal symbol

Directly Derives

If there is a production in the form of $N \rightarrow \gamma$ then we can directly derive $\alpha N \beta \rightarrow \alpha \gamma \beta$, where α and β are possibly empty sequences of terminal and nonterminal symbols.

Derives

Given a sequence of terminal and nonterminal symbols, α , derives a sequence β , written $\alpha \Rightarrow^* \beta$ if there is a finite sequence of zero or more direct derivation steps that start from α and finishing with β , there must be one or more sequence $\gamma_1, \gamma_2, \dots, \gamma_n$ such that $\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n = \beta$. Note that zero steps are allowed.

Nullable

A possibly empty sequence of symbols, α , is nullable if $\alpha \Rightarrow^* \epsilon$ or $\alpha \Rightarrow^*$ Nullable rules

- *isnullable*

- any terminal symbol is not nullable
- a sequence of symbols is nullable if all of its constructs are nullable
- a set of alternatives is nullable if any of its constructs are nullable
- EBNF constructs for optionals and repetitions are nullable
- a nonterminal is nullable if there is a production with a nullable right-hand side

Language

The formal language $\mathcal{L}(G)$ corresponding to a Grammar, G , is:

$$\mathcal{L}(G) = \{ t \in \text{seq} \mid S \Rightarrow^* t \}$$

where S is the start symbol of G and Σ is its set of terminal symbols.

Sentences and Sentential Form

A sequence of terminal symbols t such that $S \Rightarrow^* t$ is called a *sentence* of the language.

Ambiguous grammars for sequence

A grammar, G , is ambiguous for a sentence, t , in \mathcal{L} , if there is more than one parse tree for tree for t .

Ambiguous grammars

A grammar, G , is ambiguous if it is ambiguous for any sentence in $\mathcal{L}(G)$.

Left and right associative operators

To remove the ambiguity and treat $"-"$ as a left associate operator (as usual) we can rewrite the grammar to

$$E \rightarrow E "-" T$$
$$E \rightarrow T$$
$$T \rightarrow N$$

and to treat $"-"$ as right associative we use

$$E \rightarrow T "-" E$$
$$E \rightarrow T$$
$$T \rightarrow N$$

Dangling Else

Dangling else is a problem that arises when we have an if-then-else statement and the else is ambiguous as to which if it belongs to.

Recursive-Descent Parsing

Backus-Naur Form (BNF)

Allows a context-free grammar to be described by a set of productions of the form $N \rightarrow \alpha$ Where N is a nonterminal symbol and α is a (possibly empty) sequence of terminal and nonterminal symbols

The set of productions:

$$N \rightarrow \alpha_1$$
$$N \rightarrow \alpha_2$$
$$\dots$$
$$N \rightarrow \alpha_n$$

can be written as $N \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$

Extended BNF (EBNF)

Extends BNF to allow: An *optional* syntactic construct [in square brackets] $\textit{RelCondition} \rightarrow \textit{Exp} [\textit{RelOp Exp}]$ A *repetition* construct in curly braces $\textit{StatementList} \rightarrow \textit{Statement} \{ \textit{SEMICOLON Statement} \}$ The extended notation does not add any expressive power - any EBNF grammar can be converted to a BNF grammar...

Translating EBNF to BNF

Replace optional construct $[S]$ by a new nonterminal \textit{OptS} :

$$\textit{OptS} \rightarrow S | \epsilon$$

For example, we can rewrite

$$\textit{RelCondition} \rightarrow \textit{Exp} [\textit{RelOp Exp}]$$
 as
$$\textit{RelCondition} \rightarrow \textit{Exp} \textit{OptRelExp}$$
$$\textit{OptRelExp} \rightarrow \textit{RelOp Exp} | \epsilon$$

Replace grouping construct (S) by a new nonterminal \textit{GrpS} : $\textit{GrpS} \rightarrow S$

For example, we can rewrite $\textit{RepF} \rightarrow (\textit{TIMES} | \textit{DIVIDE}) \textit{Factor RepF} | \epsilon$ as

$$\textit{RepF} \rightarrow \textit{TermOp Factor RepF} | \epsilon$$
$$\textit{TermOp} \rightarrow \textit{TIMES} | \textit{DIVIDE}$$

Recursive-descent parsing

A recursive-descent parser is a recursive program to recognise sentences in a language:

- Input is a stream of lexical tokens, represented by tokens of type `TokenStream`.
- Each nonterminal symbol N has a method *parseN* that:

- recognises the longest string of lexical tokens in the input stream, starting from the current token, derivable from N ;
- as it parses the input it moves the current location forward;
- when it has finished parsing N , the current token is the token immediately following the last token matched as part of N (which may be at the end-of-file token).

matching a list of alternatives: example

```
Factor →
LPAREN RelCondition RPAREN |
NUMBER |
LValue

void parseFactor () {
    if (tokens.isMatch(Token.LPAREN)) {
        tokens.match(Token.LPAREN);
        parseRelCondition();
        tokens.match(Token.RPAREN);
    } else if (tokens.isMatch(Token.NUMBER)) {
        tokens.match(Token.NUMBER);
    } else if (tokens.isMatch(Token.IDENTIFIER)) {
        parseLValue();
    } else {
        errors.error("Syntax-error");
    }
}
```

matching an optional construct: example

$$\textit{RelCondition} \rightarrow \textit{Exp} [\textit{RelOp Exp}]$$

```
void parseRelCondition () {
    parseExp();
    if (tokens.isIn(REL_OPS_SET)) {
        parseRelOp();
        parseExp();
    }
}
```

matching a repetition construct: example

$$\textit{Term} \rightarrow \textit{Factor} (\textit{TIMES} | \textit{DIVIDE}) \textit{Factor}$$

```
void parseTerm () {
    parseFactor();
    while (tokens.isIn(TERM_OPS_SET)) {
        if (tokens.isMatch(Token.TIMES)) {
            tokens.match(Token.TIMES);
        } else if (tokens.isMatch(Token.DIVIDE)) {
            tokens.match(Token.DIVIDE);
        } else {
            fatal("Internal-Error");
        }
    }
}
```

```

    // unreachable
    }
    parseFactor();
}
}

```

recovery strategy for matching a single token

```

void parseWhileStatement() {
    tokens.match(Token.KW _WHILE);
    parseCondition();
    tokens.match(Token.KW _DO,
        STATEMENT.START.SET);
    parseStatement();
}

```

Generally, it is

```
tokens.match(expected, follows)
```

maybe add parsing recovery sets here?

LL(1) grammars

We indicated above that there are 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 Left to right, the second “L” refers to the fact that they produce a Leftmost derivation sequence, and the “1” indicates that their parsers use one symbol lookahead (i.e., the current token is the lookahead).

Definition 1 (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, $\text{First}(N)$ and $\text{Follow}(N)$ are disjoint

First & Follow Sets

First Sets

The first set for a construct α records

- the set of terminal symbols α can start with, and
- if α is nullable, it contains the empty string ϵ to indicate that.

Calculating First Sets

Let

- α be a terminal symbol,
- $\alpha_1, \alpha_2, \dots, \alpha_n$ be strings of (terminal and nonterminal) symbols, and
- A be a nonterminal symbol defined by a single production

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n,$$

then

$$\text{Fst}(\epsilon) = \{\epsilon\}$$

$$\text{Fst}(\alpha) = \{\alpha\}$$

$$\text{Fst}(\alpha_1 | \alpha_2 | \dots | \alpha_n) = \text{Fst}(\alpha_1) \cup \text{Fst}(\alpha_2) \cup \dots \cup \text{Fst}(\alpha_n)$$

$$\text{Fst}(A) = \text{Fst}(\alpha_1 | \alpha_2 | \dots | \alpha_n)$$

Let S_1, S_2, \dots, S_n be (terminal or nonterminal) symbols, then

$$\text{Fst}(S_1 S_2 \dots S_n) =$$

$$\text{Fst}(S_1) - \{\epsilon\}$$

$$\cup \text{Fst}(S_2) - \{\epsilon\} \quad \text{if } S_1 \text{ is nullable}$$

$$\cup \dots$$

$$\cup \text{Fst}(S_i) - \{\epsilon\} \quad \text{if } S_1 - S_{i-1} \text{ is nullable}$$

$$\cup \dots$$

$$\cup \text{Fst}(S_n) - \{\epsilon\} \quad \text{if } S_1 - S_{n-1} \text{ is nullable}$$

$$\cup \{\epsilon\} \quad \text{if } S_1 - S_n \text{ is nullable}$$

Calculating Algorithmically

Start with the first sets for all nonterminals being the empty set and note that the first set for every terminal symbol, α , is the singleton set $\{\alpha\}$. We then make a pass over all production in a grammar considering all alternatives and process as follows.

- If there is a production of the form $N \leftarrow \epsilon$, we add ϵ to the first set of N .
- If there is a production of the form $N \leftarrow S_1 S_2 \dots S_n$, then for each $i \in 1..n$, if for all $j \in 1..i-1$, S_j is nullable, we add the current first set for S_i minus ϵ to the first set for N .
- If every construct S_1, \dots, S_n is nullable, we add ϵ to the first set for N .

We repeat the the passes until no set is modified in the pass, in which case we are finished.

Example

$$A \leftarrow B x | C \quad (1)$$

$$B \leftarrow C y | D \quad (2)$$

$$C \leftarrow D z | \epsilon \quad (3)$$

$$D \leftarrow A w \quad (4)$$

A	$\{\}$	$\{\epsilon\}$	$\{\epsilon, y\}$	$\{\epsilon, y, w\}$	$\{\epsilon, y, w\}$
B	$\{\}$	$\{y\}$	$\{y\}$	$\{y, w\}$	$\{y, w\}$
C	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon, w\}$	$\{\epsilon, w, y\}$	$\{\epsilon, w, y\}$
D	$\{\}$	$\{w\}$	$\{w, y\}$	$\{w, y\}$	$\{w, y\}$

Follow Sets