COMP4403 Crib Sheet

Parsing Theory

Context Free Grammars

Basic Example of Context Free Grammar
$E \rightarrow E\ Op\ E$
$E \rightarrow$ "("E")"
$E \rightarrow number$
$Op \rightarrow$ " + "
$Op \rightarrow$ " − "
$Op \rightarrow$ " ∗ "
Has start symbol E, nonterminals $\{E, Op\}$, and terminals
$\{\backslash(", \backslash)", number, \backslash + ", \backslash - ", \backslash * "\}$
A context-free grammar consists of:

- A finite set, $\sum$, of terminal symbols.
- A finite nonempty set of nonterminal symbols (disjoint from the terminal symbols).
- A finite nonempty set of producions 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 of 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 $\alpha \stackrel{*}{\Rightarrow} \beta$ if there is a finitie sequence of zero or more direct derivation steps that start from α and finishing with β, there must be one or more sequence $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that $\alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_n = \beta$. Note that zero steps are allowed.

Nullable

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

- $\epsilon isnullable$
- 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:
$\mathcal{L}(G) = \{t \in seq \sum | S \stackrel{*}{\Rightarrow} t\}$
where S is the start symbol of G and $\sum$ is its set of terminal symbols.

Sentences and Sentenial Form

A sequence of terminal symbols t such that $S \stackrel{*}{\Rightarrow} t$ 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 setence in $\mathcal{L}(G)$.

Left and right associative operators

To remove the ambiguit 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.

Recusive-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:
$N \rightarrow \alpha_1$
$N \rightarrow \alpha_2$
$\ldots$
$N \rightarrow \alpha_n$
can be wrriten as $N \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$

Extended BNF (EBNF)

Extends BNF to allow: An *optional* syntatic construct [in square brackets] $RelCondition \rightarrow Exp\ [RelOp\ Exp]$ A *reptition* construct in curly braces $StatementList \rightarrow Statement\ \{SEMICOLON\ Statement\}$
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 $OptS$:
$OptS \rightarrow S | \epsilon$
For example, we can rewrite $RelCondition \rightarrow Exp\ [RelOp\ Exp]$ as
$RelCondition \rightarrow Exp\ OptRelExp$
$OptRelExp \rightarrow RelOp\ Exp | \epsilon$
Replace grouping construct (S) by a new nonterminal $GrpS$: $GrpS \rightarrow S$
For example, we can rewrite $RepF \rightarrow (TIMES \mid DIVIDE)\ Factor\ RepF\ | \epsilon$ as
$RepF \rightarrow TermOp\ Factor\ RepF\ | \epsilon$
$TermOp \rightarrow TIMES \mid DIVIDE$

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 lexcial 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

$Factor \rightarrow$
$LPAREN\ RelCondition\ RPAREN\ |$
$NUMBER\ |$
$LValue$

```
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

$RelCondition \rightarrow\ Exp\ [\ RelOp\ Exp\ ]$

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

matching a repetition construct: example

$Term \rightarrow\ Factor\ (\ TIMES \mid DIVIDE\ )\ Factor$

```
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 gram- mars to ensure they are suitable for recursive-descent predictive parsing. The class of grammars that are suitable is re- ferred 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 se- quence, 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, First(N) and Follow(N) are disjoint