COMP4403 Crib Sheet

Parsing Theory

Context Free Grammars

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 $A \to \alpha$, 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

Nullable

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

Left and right associative operators

To remove the ambiguit and treat "-" as a left associate operator (as usual) we can rewrite the grammar to
$E \to E - T$
$E \to T$
$T \to N$
and to treat "-" as right associative we use
$E \to T - E$
$E \to T$
$T \to N$

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

Translating EBNF to BNF

Replace optional construct $[S]$ by a new nonterminal $OptS$:
$OptS \to S|\epsilon$
For example, we can rewrite
$RelCondition \to Exp\ [RelOp\ Exp]$
as
$RelCondition \to Exp\ OptRelExp$
$OptRelExp \to RelOp\ Exp|\epsilon$

Replace grouping construct (S) by a new nonterminal $GrpS$:
$GrpS \to S$
For example, we can rewrite
$RepF \to$
$(TIMES \mid DIVIDE)\ Factor\ RepF\ |\epsilon$
as
$RepF \to TermOp\ Factor\ RepF\ |\epsilon$
$TermOp \to TIMES \mid DIVIDE$

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).

parsing example

Write parsing example here

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

$$A \to \alpha_1|\alpha_2|\ldots|\alpha_n,$$

then

$$\begin{aligned} Fst(\epsilon) &= \{\epsilon\} \\ Fst(\alpha) &= \{\alpha\} \\ Fst(\alpha_1|\alpha_2|\ldots|\alpha_n) &= Fst(\alpha_1) \cup Fst(\alpha_2) \\ &\quad \cup \cdots \cup Fst(\alpha_n) \\ Fst(A) &= Fst(\alpha_1|\alpha_2|\ldots|\alpha_n) \end{aligned}$$

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

$$\begin{aligned} &Fst(S_1 S_2 \ldots S_n) = \\ &\quad Fst(S_1) - \{\epsilon\} \\ &\quad \cup Fst(S_2) - \{\epsilon\} && \text{if } S_1 \text{ is nullable} \\ &\quad \cup \ldots \\ &\quad \cup Fst(S_i) - \{\epsilon\} && \text{if } S_1\text{-}S_{i-1} \text{ is nullable} \\ &\quad \cup \ldots \\ &\quad \cup Fst(S_n) - \{\epsilon\} && \text{if } S_1\text{-}S_{n-1} \text{ is nullable} \\ &\quad \cup \{\epsilon\} && \text{if } S_1\text{-}S_n \text{ is nullbale} \end{aligned}$$

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 grammer considering all atlternatives and process as follows.
- If there is a production of the form $N \to \epsilon$, we add ϵ to the first set of N.
- If there is a production of the form $N \to S_1 S_2 \ldots S_n$, then for each $i \in 1..n$, if for all $j \in 1..i-1$, S_j is nullabe, we add the current first set for S_i minus ϵ to the first set for N.
- If every construct $S_1, ..., S_n$ is nullabe, 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

$$A \to B\,x|C$$
$$B \to C\,y|D$$
$$C \to D\,z|\epsilon$$
$$D \to A\,w$$

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

The follow set for a nonterminal, N, is the set of terminal symbols that may follow N in any context within the grammar. End-of-file is represented by the special terminal symbol \$ in which a follows N.
A nonterminal, N, is followed by a terminal symbol, a, if there is a derivation from $S\$$

Calculating Follow Sets

We compute the Follow set for a nonterminal, N, using two rules.
- If there is a production of the form
  $$A \to \alpha N \beta$$
  then any symbols that can start β can follow N, and hence $Follow(N)$ must include all the terminal symbols in $First(\beta)$. Note that ϵ is not included even if it appears in $First(\beta)$.
  $$First(\beta) - \{\epsilon\} \subseteq Follow(N)$$
- If there is a production of the form
  $$A \to \alpha N \beta$$
  and β is nullable, then any token that can follow A can also follow N. Hence,
  $$Follow(A) \subseteq Follow(N)$$
  The case where β is nullable includes the case when β is empty and the production is of the form
  $$A \to \alpha N$$

Calculating Algorithmically

Start with all nonterminal symbols having an empty follow set, {}, except for the start symbol, S, which has the follow set {\$}.
We make a pass through the grammar examining the right side of every production. For each occurence of a nonterminal within the right side of some production, we augment the Follow set for that nonterminal according to the following process. Assume we are processing an occurence of N and the production is of the form
$$A \to \alpha N \beta$$

we add $First(\beta) - \{\epsilon\}$ to the Follow set computer for N so far, and if β is nullable, we also add the current Follow set for A to the Follow set for N.
After making a complete pass, we repeat the process with the Follow sets computed so far until no Follow sets are modified, in which case we are done.

Example

$$S \to xAB \qquad Fst(S) = \{x\}$$
$$A \to y|zB \qquad Fst(A) = \{y, z\}$$
$$B \to \epsilon|Ax \qquad Fst(B) = \{\epsilon, y, z\}$$

S	{\$}	{\$}	{\$}	{\$}
A	{}	$\{y, z, \$, x\}$	$\{y, z, \$, x\}$	$\{y, z, \$, x\}$
B	{}	$\{\$, y, z\}$	$\{\$, y, z, x\}$	$\{\$, y, z, x\}$

LL(1) Grammar

A BNF grammer is LL(1) if for each nonterminal, N, wher $N \to \alpha_1|\alpha_2|\ldots|\alpha_n$,
- the First sets for each pair of alternatives for N are disjoint, and
- if N is nullable, $First(N)$ and $Follow(N)$ are disjoin.

Bottom Up Parsing

Shift/Reduce Parsing

Makes use of a parse stack and has three actions:
- **shift** - push the next input symbol onto the stack,
- **reduce** - if a sequence of symbols on the top of the stack, α, matches the right side of some production $N \to \alpha$, then the sequence α on top of the stack is replaced by N.
- **accept** - if the stack contains just the start symbol and there is no input left, the input has been recognised and is accepted.

LR(0) Grammars

An LR(0) parsing item is of the form
$$N \to \alpha \bullet \beta$$
which indicates that, in matching N, α has been matched and β is yet to be matched where
- N is a nonterminal symbol,
- α and β are possibly empty sequences of (terminal and nonterminal) symbols such that $N \to \alpha\ \beta$ is a production of the grammer, and
- $\bullet$ marks the current position in matching the right side.

Parsing Automaton

An LR(0) parsing automaton consists of
- a finite set of states, each of which consists of a set of LR(0) parsing items, and
- transitions between states, each of which is labelled by a transition symbol.

Each state in an LR(0) parsing automaton must have only one associated parsing action.

Automaton States

If a state has an LR(0) item of the form
$$N \to \alpha \bullet M\beta$$
where the nonterminal M has productions
$$M \to \alpha_1$$
$$M \to \alpha_2$$
$$\vdots$$
$$M \to \alpha_m$$
then the state also includes the derived items
$$M \to \bullet\alpha_1$$
$$M \to \bullet\alpha_2$$
$$\vdots$$
$$M \to \bullet\alpha_m$$

Goto States

If a state s_i has an LR(0) item of the form
$$N \to \alpha \bullet x\beta$$
where x is either a terminal symbol or a nonterminal symbol, then there is a goto state, s_j, from the state s_1 on x, and s_j includes a kernel item of the form
$$N \to \alpha x \bullet \beta$$
If there are multiple items in s_i with the same x immediately to the right of the $\bullet$ then the goto state s_j includes all those items but with the $\bullet$ after that occurence of x rather than before it.

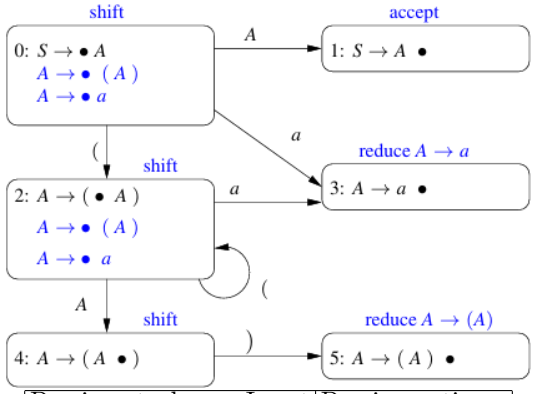
Actions

An LR(0) item of the form
- $N \to \alpha \bullet a\beta$ where a is a terminal symbol, indicates the state containing the item has a shift **parsing** action
- $S' \to S\bullet$ where S' is the (introduced) start symbol for the grammer, inidcates the state containing the item has an **accept** action
- $N \to \alpha\bullet$ where N is not the (introduced) start symbol for the grammar, indicates the state containing the item has a parsing action **reduce** $N \to \alpha$

A shift action at end-of-file is an error, as is an accept action when the input is not at end-of-file.

Example

$$S \to A \qquad A \to (\ A\ ) \qquad A \to a$$



Parsing stack	Input	Parsing action
\$0	((a))\$	shift
\$0(2	(a))\$	shift
\$0(2(2	a))\$	shift
\$0(2(2a3	))\$	reduce $A \to a$
\$0(2(2A4	))\$	shift
\$0(2(2A4)5	)\$	reduce $A \to (A)$
\$0(2A4	)\$	shift
\$0(2A4)5	\$	reduce $A \to (A)$
\$0A1	\$	accept

Parsing Conflicts

If a state in an LR(0) parsing automaton has more than one action, there is a parsing action conflict.
A grammar is LR(0) if none of the states in its LR(0) parsing automaton contains a parsing action conflict.

LR(1) Grammars

An LR(1) parsing item is a pair
$$[N \to \alpha \bullet \beta, T]$$
consisting of
- an LR(0) parsing item $N \to a \bullet \beta$, and
- a set T of terminal symbols called a look-ahead set.

The above item indicates that in matchin N, α has been matched and β is yet to be matched, in a context in which N can be followed by a terminal symbol in the set T.

Parsing Automaton

The kernal item of the initial state is
$$[S' \to \bullet S, \$]$$
where S is the start symbol of the grammar and we introduce a fresh replacement start symbol S' and production $S' \to S$. This fresh production is used to determine when parsing has completed.

Derived Items

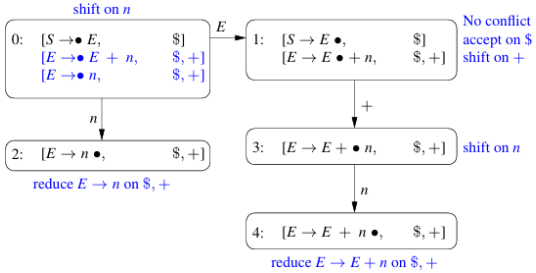
If a state has an LR(1) item of the form
$$[N \to \alpha \bullet M\beta, T]$$
where the nonterminal M has productions
$$M \to \alpha_1|\alpha_2|\ldots|\alpha_m$$
and $T = \{a_1, a_2, \ldots, a_n\}$, then the state also includes the derived items
$$[N \to \bullet\alpha_1, T]\ldots[N \to \bullet\alpha_m, T]$$
where if β is not nullable $T' = First(\beta)$ and if β is nullable, $T' = First(\beta) - \{\epsilon\} \cup T$.

Parsing Actions

An LR(1) item of the form
- $[N \to \alpha \bullet a\beta, T]$, where a is a terminal symbol, indicates the state containing the item has a **shift** parsion action if the next input x is a
- $[S' \to \S\bullet, \$]$, where S' is the added start symbol for the grammar, indicates the state containing the item has an **accept** action if there is no more input.
- $[N \to \alpha\bullet, T]$, where N is not S', indicates the state containing the item has a parsing action **reduce** $N \to \alpha$ if the next input x is in T.

Example

$$S \to E \qquad E \to E + n \qquad E \to n$$



Parsing Action Conflicts

If a state in an LR(1) parsing automaton has more than one action for a look-ahead terminal symbol, there is a parsing action conflict.
A grammar is LR(1) if none of the states in its LR(1) parsing automaton contains a parsing action conflict.

LALR(1) Parsing

- Look-Ahead merged LR(1) parsing where each state of an LALR(1) parsing automaton consists of a set of LR(1) items.
- An LALR(1) parsing automaton can be formed from an LR(1) parsing automaton by merging states that have identical sets of LR(0) items but possibly different look-ahead sets.
- Each item in the merged state consists of one of the LR(0) items in the states being merged with a look-ahead set that is the union of the look-ahead sets of that item in all the states being merged.
- The parsing actions for LALR(1) parsing are defined in the same way as for LR(1) parsing.
- A grammar is LALR(1) if none of the states in its LALR(1) parsing automaton contains a parsing action conflict.

Left Factoring & Left Recursion

Left Factoring Productions

Not all EBNF grammars are suitable for Recursive-Descent Parsing, however, sometimes we can rewrite them into a form that is suitable.

Left Factor Rewriting Rule

To remove the left factor from

$$A \to \alpha\ \beta \mid \alpha\ \gamma,$$

we can rewrite the production using an aditional nonterminal A' as

$$A \to \alpha\ A'$$
$$A' \to \beta \mid \gamma$$

Left Recursive Productions

A production of the form

$$E \to E\ +\ T \mid T$$

is not suitable for RDP because the left recursion in the grammar leads to an infinite recursion.

Immediate Left Recursion Rewriting Rule (Simple)

To remove the left recursion from

$$A \to A\ \alpha \mid \beta,$$

we can rewrite the production as

$$A \to \beta\ A'$$
$$A' \to \epsilon \mid \alpha\ A'$$

Immediate Left Recursion Rewriting Rule (General)

To remove the left recursion from the general case

$$A \to A\alpha_1|A\alpha_2|\ldots|A\alpha_n|\beta_1|\beta_2|\ldots|\beta_m,$$

we can use grouping to see the structure in the same form as the simple case

$$A \to A(\alpha_1|\alpha_2|\ldots|\alpha_n)|(\beta_1|\beta_2|\ldots|\beta_m)$$

which allows us to rewrite the production as follows,

$$A \to (\beta_1|\beta_2|\ldots|\beta_m)A'$$
$$A' \to \epsilon|(\alpha_1|\alpha_2|\ldots|\alpha_n)A'$$

Indirect Left Recursion Rewriting Rule

The following productions have indirect recursion from $A \to B \to C \to A$.

$$A \to B\ \alpha$$
$$B \to C\ \beta$$
$$C \to A\ \gamma_1 \mid \gamma_2$$

To remove the recursion we can first collapse B into A as follows.

$$A \to C\ \beta\ \alpha$$
$$C \to A\ \gamma_1 \mid \gamma_2$$

Then we can collapse C into A.

$$A \to (A\ \gamma_1 \mid \gamma_2)\ \beta\ \alpha$$
$$\downarrow$$
$$A \to A\ \gamma_1\ \beta\ \alpha \mid \gamma_2\ \beta\ \alpha$$

This now leaves a simple direct left recursion which we can remove as follows.

$$A \to \gamma_2\ \beta\ \alpha\ A'$$
$$A' \to \gamma_1\ \beta\ \alpha\ A'$$