Syntactic Analysis

- Syntactic analyzers, or parsers, analyze the structure of the program and its component statements and check for errors.
- Driver of the translation process:
  - Controls the lexical analyzer, which provides tokens in response to the parser’s requests.
  - May supervise the operation of the intermediate code generator.

Structure of Program

- Specified by context-free grammar.
- Context-free grammar cannot describe information such as types in a program. (Context-sensitive information).
- However, context-sensitive grammar is too complex and too expensive to implement.
- Context-sensitive information, instead, is processed in the semantic phase of translation.

Grammar Review

- Terminals (T) and Nonterminals (V/N)
- Production Rules (P/R)
- Start symbol (S)
- Parse Tree / Derivation
  - Ambiguous Grammars
- Leftmost and Rightmost Derivation

Review of Terms

- Nothing is usually written as \( \lambda \) (lambda).
- Derivation, parse
  - process of finding a parse tree for a given string of tokens
- Parse tree
  - pictorially shows how the start symbol of a grammar derives a string in the language
  - the tree gives meaning - tells us how to group things.

Grammar Productions

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E - E \\
E & \rightarrow E \times E \\
E & \rightarrow E / E \\
E & \rightarrow ( E ) \\
E & \rightarrow a | b | c ...
\end{align*}
\]

Parse Tree #1

```
a - b * c
```

```
E -> E + E
  /   |
 /     |
E-> E - E
  |
  |
E -> E * E
  |
  |
E -> E / E
  |
  |
E -> ( E )
  |
  |
E-> a | b | c ...
```
Issues to Consider

- Precedence of Operators
- Ambiguity
  - Can you generate 2 or more parse trees for a given string?

Revised Grammar

- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow F$
- $F \rightarrow a \mid b \mid c \ldots$
- $F \rightarrow (E)$

Left vs. Right Derivation

- **Left Derivation**
  - ALWAYS expand the leftmost possible non-terminal first.
  - Top-Down
- **Right Derivation**
  - ALWAYS expand the rightmost possible non-terminal first.
  - Bottom-Up

Parsing

- **Top-down**
  - Efficient parsers can be constructed easily by hand using top-down
- **Bottom-up**
  - Can handle a larger class of grammars and translation schemes
  - Software tools for generating parsers directly from grammars use bottom-up

Grammar Manipulations

- Some grammar productions cannot be used directly in constructing certain types of parsers.
- They must be rewritten.
  - We want to do various manipulations in such a way that the set of sentences recognized is not changed.
Grammar Manipulations

- Remove useless non-terminals.
- Rewrite the grammar to remove ambiguity.
- Rewrite the grammar to remove left recursion.
- Left factoring the grammar.
- Remove lambda-productions.
- Remove unit-productions.

Useless non-terminals

\[
\begin{align*}
S &\rightarrow A \mid B \\
A &\rightarrow a \mid aA \\
B &\rightarrow Bb \\
C &\rightarrow c
\end{align*}
\]

- B & C are useless because they will never be used to derive a sentence.

Revised Grammar:

\[
\begin{align*}
S &\rightarrow A \\
A &\rightarrow a \mid aA
\end{align*}
\]

Rewrite to remove ambiguity

- The easiest way to do this is to add new non-terminals.
  - Example: \( E \rightarrow E + E \mid E * E \ldots \) becomes \( E \rightarrow E + T \mid T * F \ldots \)
- Note: It doesn’t change the grammar at all. Either way the grammar will still recognize the sentence \((a - b * c)\) but the meaning of the sentence will be different. Also certain parsers don’t know how to deal with too many choices.

Remove left recursion

- Our first parsing method will require us to remove left recursion.
- There are 2 forms of left-recursion:
  - Direct
  - Indirect

Direct vs. Indirect Left Recursion

- Direct
  - \( A \rightarrow A B \)
  - where \( B \) is anything else
- Indirect
  - \( A \rightarrow BC \)
  - \( B \rightarrow AD \)
  - where \( C \) & \( D \) are anything else

Removal Procedure

- The idea is as follows.
- For each production:
  \( A \rightarrow A_B \mid A_C \mid X \mid Y \)
- We look at it from the very left instead of right.
- The LEFTMOST thing that \( A \) matches must be something without recursion, namely \( X \) & \( Y \).
Removal Procedure (cont.)

- What comes to the right of this must be something that can come after an A.
- So we introduce a new symbol B to represent that which can come after an A.
  \[ A \rightarrow xB \mid yB \]
  \[ B \rightarrow bB \mid cB \mid \lambda \]
- Anything that can be produced by the former can now be produced by the new grammar.

Example

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
E' & \rightarrow +T E' \mid \lambda \\
T & \rightarrow T * F \mid F \\
T' & \rightarrow *F T' \mid \lambda \\
F & \rightarrow (E) \mid id \\
\end{align*}
\]

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing (top-down)
- The basic idea is that when it is not clear which of the two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make a choice.

Example

\[
\begin{align*}
Stmt & \rightarrow \text{if} \ Expr \ \text{then} \ Stmt \ \text{else} \ Stmt \\
Stmt & \rightarrow \text{if} \ Expr \ \text{then} \ Stmt \\
\end{align*}
\]

- On seeing the input token if, we cannot immediately tell which production to choose to expand Stmt.

Left Factoring (cont.)

In general, if we have this production:
\[ A \rightarrow xB \mid xC \]
we do not know whether to expand A to xB or to xC.
However, we can defer the decision by expanding A to a \( A' \).
Then after seeing the input derived from x, we expand \( A' \) to B or to C.

Example

\[
\begin{align*}
A & \rightarrow xB \mid xC \\
\end{align*}
\]

becomes:
\[
\begin{align*}
A & \rightarrow xA' \\
A' & \rightarrow B \mid C \\
\end{align*}
\]
Example:

Stmt --> if Expr then Stmt else Stmt | if Expr then Stmt

Stmt --> if Expr then Stmt S’
S’ --> else Stmt |

Question: Is this new grammar still ambiguous?

Parsers & Recognizers

- Assume we know that the grammar is unambiguous.
- Given an input string as a sequence of tokens, we can ask:
  - “Is this input syntactically valid?” or
  - “Can it be generated from the grammar?”
- ARecognizer is an algorithm that does this Boolean-value test.

Parsers and Recognizers (cont.)

- We can also require more of the algorithm and ask:
  - “Is this input valid, and if it is, what is its structure (parse tree)?
- Parser is an algorithm that answers this more general question.

General Approaches to Parsing

- Top-Down
  - recursive decent technique
  - it discovers the parse tree corresponding to a token sequence by starting at the top of the tree (the start symbol) and then expanding it (via predictions) in a depth-first manner.
  - “Predictive” - always predict the rule that is to be matched before matching actually begins.

General Approaches to Parsing

- Bottom-Up
  - it discovers the structure of a parse tree by beginning at its bottom (the leaves of the tree, which are terminal symbols) and determining the rules used to generate the leaves.
  - then the rules used to generate the immediate parents of the leaves are discovered.
  - the parser continues until it reaches the rule used to expand the start symbol.

Top-Down vs. Bottom-Up

- Example:
  Program --> begin Stmts end
  Stmts --> Stmt; Stmts
  Stmts --> begin Stmts end
  Stmt --> SimpleStmt
  Stmt --> begin Stmts end
Top-down Parsing

\[
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow T * F \\
T \rightarrow F \\
F \rightarrow \text{id} \\
F \rightarrow (E)
\]

Top-Down Parsing

Let us start by taking the grammar for expressions (without left-recursion)

\[
E \rightarrow T + E \\
E \rightarrow T \\
T \rightarrow F * T \\
T \rightarrow F \\
F \rightarrow \text{id} \\
F \rightarrow (E)
\]

Construct a parser

- Suppose we have a routine `nextToken()` that would return the next token, plus routines `currentPosition()` which would return the current input position, and `resetPosition(i)` which would set the input back to position i.
- Using these, we can construct a parser.

Construct a parser

- Our goal is to recognize an E, so we start by writing a procedure which will return true if the input is something that matches one of the two rules for E, namely:
  \[
  E \rightarrow T + E \\
  E \rightarrow T
  \]
  - So we simply try each of them in turn. If the first fails, we try the second.
  - In order to back up, we need to reset things to the point they were when we tried the first alternative.
boolean success; // set true if we are successful
integer startingPosition; // where we were at the start
procedure E()
    startingPosition = currentPosition();
    success = false;
    if T() then
        if match(PLUS) then if E() then success = true;
        if not success then /* try next alternative */
            resetPosition(startingPosition);
        if T() then success = true;
    return success;

Construct a parser
- What happens if we had used the original grammar?
- This technique will work for ANY context-free language that does not have left recursion. One of the most general parsing techniques.
- What problems does it have? It backs up a lot.
- Backtracking. We would like a way to avoid backtracking.
- A parser that does not use backtracking is known as a **predictive parser**.

Backtracking
- Why do we need backtracking?
  - When faced with a choice, we didn’t know which one to select.
- But we know that we can rewrite our grammar so we did know this.
- Left-factoring

Left-factoring grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rewritten grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>E --&gt; T E'</td>
<td>E --&gt; T E'</td>
</tr>
<tr>
<td>E' --&gt; + T E'</td>
<td>E' --&gt; + E</td>
</tr>
<tr>
<td>T --&gt; F T'</td>
<td>T --&gt; F T'</td>
</tr>
<tr>
<td>T' --&gt; * T</td>
<td>T' --&gt; * T</td>
</tr>
<tr>
<td>F --&gt; id</td>
<td>F --&gt; id</td>
</tr>
<tr>
<td>F --&gt; (E)</td>
<td>F --&gt; (E)</td>
</tr>
</tbody>
</table>

Left-factoring vs remove left-recursion

<table>
<thead>
<tr>
<th>Remove left-recursion from original grammar</th>
<th>Left-factoring reversed grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>E --&gt; T E'</td>
<td>E --&gt; T E'</td>
</tr>
<tr>
<td>E' --&gt; + T E'</td>
<td>E' --&gt; + E</td>
</tr>
<tr>
<td>T --&gt; F T'</td>
<td>T --&gt; F T'</td>
</tr>
<tr>
<td>T' --&gt; * T</td>
<td>T' --&gt; * T</td>
</tr>
<tr>
<td>F --&gt; id</td>
<td>F --&gt; id</td>
</tr>
<tr>
<td>F --&gt; (E)</td>
<td>F --&gt; (E)</td>
</tr>
</tbody>
</table>

procedure E()
    if T() then
        if E'() then return true;
    error();

procedure E'()
    if match(PLUS) then
        if T() then if E'() then return true;
    error();
    else return true;

procedure F()
    if match(ID) then return true;
    if match(’’) then if E() then if match(’’) then return true;
Recursive Descent Parser

- This technique is called Recursive Descent.
- Many people regard it as the method of choice if you are writing a parser by hand.
- It is easy to write directly from the grammar, you have reasonably good error messages.
- Problem - the programs are BIG.

LL & LR

- LL & LR are used to describe the type of parsers.
- The token sequence will be parsed from left to right.
- Leftmost or Rightmost Derivation Parse

LL & LR

- LL & LR parsers are usually associated with a number.
- LL(n) - where n is the number of look ahead symbols (tokens, not just input characters)
- LL(1) and LR(1) are the most common parsers.

LL (1)

- A grammar is said to be LL(1) if, when faced with a choice of several alternatives for the current goal, we can decide using only the NEXT token, which alternative to take.
- After left-factoring, our grammar is now LL(1).

First and Follow Sets

- Given a grammar, we can compute the first and follow sets for the grammar.
- First Set:
  - The first token you can find when you are looking for an A.
- Follow Set:
  - The next token you can find after you have seen an A.