Syntax Analysis

Chapter 4, Part I
Context Free Grammar

The Structure of a Compiler

The Functionality of the Parser

• **Input:** sequence of tokens from lexer
• **Output:** parse tree of the program

Parser

• Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
• Determines if the input is syntactically well formed
• Guides checking at deeper levels than syntax
• Builds an IR representation of the code

Example

• Source input
  \[
  \text{if } x = y \text{ then } 1 \text{ else } 2 \text{ fi}
  \]
• Parser input
  \[\text{IF ID = ID THEN INTELSE INT FI}\]
• Parser output
  \[
  \begin{align*}
  \text{IF-THEN-ELSE} \\
  = & \quad \text{INT} \\
  \text{ID} & \quad \text{ID} \\
  \end{align*}
  \]

Comparison with Lexical Analysis

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexer</td>
<td>Sequence of characters</td>
<td>Sequence of tokens</td>
</tr>
<tr>
<td>Parser</td>
<td>Sequence of tokens</td>
<td>Parse tree</td>
</tr>
</tbody>
</table>
The Study of Parsing

The process of discovering a derivation for some sentence

- Need a mathematical model of syntax — a grammar \( G \)
- Need an algorithm for testing membership in \( L(G) \)
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap for our study of parsing

1. Context-free grammars and derivations
2. Top-down parsing
   - Generated LL(1) parsers
3. Bottom-up parsing
   - Generated LR(1) parsers

Context Free Grammars

- A CFG consists of
  - A set of terminals \( T \) (received from LEX)
  - A set of non-terminals \( N \)
  - A start symbol \( S \) (a non-terminal)
  - A set of productions
    - \( X \rightarrow \varepsilon \), or
    - \( X \rightarrow Y_1 Y_2 ... Y_n \) where \( Y_i \in N \cup T \)
    - \( X \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k \)

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

Examples of CFGs (cont.)

Simple arithmetic expressions:

\[
E \rightarrow E \ast E \\
| \quad E + E \\
| \quad (E) \\
| \quad id
\]

The Language of a CFG

Read productions as replacement rules:

\[
X \rightarrow Y_1 ... Y_n \\
\text{Means } X \text{ can be replaced by } Y_1 ... Y_n \\
X \rightarrow \varepsilon \\
\text{Means } X \text{ can be erased (replaced with empty string)}
\]
Key Idea

1. Begin with a string consisting of the start symbol “S”
2. Replace any non-terminal X in the string by a right-hand side of some production $X \rightarrow Y_1 \ldots Y_n$
3. Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

More formally, write

$$X_1 \ldots X_i X_{i+1} \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

if there is a production $X_i \rightarrow Y_1 \ldots Y_m$

The Language of a CFG (Cont.)

Write

$$X_1 \ldots X_n \rightarrow Y_1 \ldots Y_m$$

if

$$X_1 \ldots X_n \rightarrow \ldots \rightarrow Y_1 \ldots Y_m$$

in 0 or more steps

The Language of a CFG

Let $G$ be a context-free grammar with start symbol $S$. Then the language of $G$ is:

$$\{ a_1 \ldots a_n | S \rightarrow^* a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \}$$

Terminals

- Terminals are called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E+E | E*E | (E) | id$$

Some elements of the language:

$$id \quad id + id$$

$$(id) \quad id * id$$

$$(id) * id \quad id * (id)$$
Notes

- Membership in a language is “yes” or “no”
  - we also need parse tree of the input
- Must handle errors gracefully
- Need an implementation of CFG’s (e.g., bison)

More Notes

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar

Derivations and Parse Trees

A derivation is a sequence of productions
\[ S \rightarrow \ldots \rightarrow \ldots \]

A derivation can be drawn as a tree
- Start symbol is the tree’s root
- For a production \( X \rightarrow Y_1 \ldots Y_n \), add children \( Y_1, \ldots, Y_n \) to node \( X \)

Derivation Example

- Grammar
  \[ E \rightarrow E+E \mid E \cdot E \mid (E) \mid id \]
- String
  \[ id \cdot id + id \]

Derivation Example (Cont.)

\[
\begin{align*}
E & \rightarrow E+E \\
& \rightarrow E \cdot E+E \\
& \rightarrow id \cdot E + E \\
& \rightarrow id \cdot id + E \\
& \rightarrow id \cdot id + id
\end{align*}
\]

Derivation in Detail (1)

\[
\begin{align*}
E & \rightarrow E+E \\
& \rightarrow E \cdot E \\
& \rightarrow id \cdot E \\
& \rightarrow id \cdot id
\end{align*}
\]
Derivation in Detail (2)

\[ E \rightarrow E + E \]

Derivation in Detail (3)

\[ E \rightarrow E + E \]
\[ E \rightarrow E * E + E \]

Notes on Derivations

• A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

• An in-order traversal of the leaves is the original input

• The parse tree shows the association of operations, the input string does not
Left-most and Right-most Derivations

- The example is a left-most derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a right-most derivation

Right-most Derivation in Detail (1)

\[ E \rightarrow E + E \]
\[ E \rightarrow E + id \]
\[ E \rightarrow E * E + id \]
\[ E \rightarrow E * id + id \]
\[ E \rightarrow id * id + id \]

Right-most Derivation in Detail (2)

\[ E \rightarrow E + E \]
\[ E \rightarrow E + id \]
\[ E \rightarrow E * E + id \]

Right-most Derivation in Detail (3)

\[ E \rightarrow E + E \]
\[ E \rightarrow E + id \]
\[ E \rightarrow E * E + id \]
\[ E \rightarrow E * id + id \]

Right-most Derivation in Detail (4)

\[ E \rightarrow E + E \]
\[ E \rightarrow E + id \]
\[ E \rightarrow E * E + id \]

Right-most Derivation in Detail (5)

\[ E \rightarrow E + E \]
\[ E \rightarrow E + id \]
\[ E \rightarrow E * E + id \]
\[ E \rightarrow E * id + id \]
Right-most Derivation in Detail (6)

- \( E \)
- \( \rightarrow E+E \)
- \( \rightarrow E+id \)
- \( \rightarrow E*E + id \)
- \( \rightarrow E*id + id \)
- \( \rightarrow id*id + id \)

Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Summary of Derivations

- We are not just interested in whether \( s \in L(G) \)
  - We need a parse tree for \( s \)
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- A grammar that produces more than one parse tree for some sentence is ambiguous
  - \( id + id * id \)

CFG v.s. RE

- Every regular language is a context-free language
  - For each state i of the NFA, create a nonterminal \( Ai \)
  - If the state i has a transition to j on input a, add the production \( Ai \rightarrow aAj \)
  - If I is an accepting state, add \( Ai \rightarrow e \)
  - If I is the start state, make \( Ai \) the start symbol
  - Ex. \( (ab)^*abb \)

CFG v.s. RE

- A context-free language is not necessary a regular language is
  - Finite automaton can’t remember # of times it has visited a particular state
  - E.g., language of balanced parentheses is not regular:
    \[ \{ (i)^j \mid i \geq 0 \} \]
Examples

Strings of balanced parentheses

Two grammars: \( \{ (i) \mid i \geq 0 \} \)

\[ S \to (S) \quad \text{OR} \quad S \to (S) \]

\[ S \to \varepsilon \quad \text{OR} \quad \varepsilon \]

Intro to Top-Down Parsing

• Terminals are seen in order of appearance in the token stream:

\[ t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \]

• The parse tree is constructed
  – From the top
  – From left to right

Recursive Descent Parsing

• Consider the grammar
  \[ E \to T + E \mid T \]
  \[ T \to (E) \mid \text{int} \mid \text{int} \ast T \]

• Token stream is: \( \text{int} \ast \text{int} \)

• Start with top-level non-terminal \( E \)

• Try the rules for \( E \) in order

Recursive Descent Parsing. Example (Cont.)

• Try \( E \to T \)

• Follow same steps as before for \( T \)
  – And succeed with \( T \to \text{int} \ast T \) and \( T \to \text{int} \)
  – With the following parse tree

Recursive Descent Parsing. Example (Cont.)

• Try \( E \to T + E \)

• Then try a rule for \( T \to (E) \)
  – But ( does not match input token \( \text{int} \)

• Try \( T \to \text{int} \)
  – Token matches.
  – But \( \ast \) after \( T \) does not match input token \( \ast \)

• Try \( T \to \text{int} \ast T \)
  – This will match but \( \ast \) after \( T \) will be unmatched

• Have exhausted the choices for \( T \)
  – Backtrack to choice for \( E \)

Recursive Descent Parsing. Notes.

• Easy to implement by hand
  – An example implementation is provided as a supplement “Recursive Descent Parsing”

• But does not always work …
Recursive-Descent Parsing

- Parsing: given a string of tokens $t_1 t_2 \ldots t_n$, find its parse tree
- Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: $t_1 t_2 \ldots t_k A \ldots$
  - Try all the productions for $A$: if $A \rightarrow BC$ is a production, the new fringe is $t_1 t_2 \ldots t_k B C \ldots$
  - Backtrack when the fringe doesn't match the string
  - Stop when there are no more non-terminals

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S a$:
  - In the process of parsing $S$ we try the above rule
  - What goes wrong?

- A left-recursive grammar has a non-terminal $S$
  - $S \rightarrow^* S \alpha$ for some $\alpha$

- Recursive descent does not work in such cases
  - It goes into a dead loop

Elimination of Left Recursion

- Consider the left-recursive grammar
  - $S \rightarrow S \alpha \mid \beta$

- $S$ generates all strings starting with a $\beta$ and followed by a number of $\alpha$

- Can rewrite using right-recursion
  - $S \rightarrow \beta S'$
  - $S' \rightarrow \alpha S' \mid \epsilon$

More Elimination of Left-Recursion

- In general
  - $S \rightarrow S \alpha_1 \mid \ldots \mid S \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

- All strings derived from $S$ start with one of $\beta_1, \ldots, \beta_m$ and continue with several instances of $\alpha_1, \ldots, \alpha_n$

- Rewrite as
  - $S \rightarrow \beta_1 S' \mid \ldots \mid \beta_m S'$
  - $S' \rightarrow \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \epsilon$

General Left Recursion

- The grammar
  - $S \rightarrow A \alpha \mid \delta$
  - $A \rightarrow S \beta$

  is also left-recursive because
  - $S \rightarrow^* S \beta \alpha$

- This left-recursion can also be eliminated
- See book, Section 4.3 for general algorithm
Summary of Recursive Descent

• Simple and general parsing strategy
  – Left-recursion must be eliminated first
  – ... but that can be done automatically
• Unpopular because of backtracking
  – Thought to be too inefficient

• In practice, backtracking is eliminated by restricting the grammar