Lecture 5 Outline

- Context free grammars
- Abstract syntax trees
- Top down parsing
- Recursive descent

Reading Assignment
Section 4.1 - 4.4

Grammars and Syntax analysis
Grammar used to derive string or construct parser
First look at deriving strings
A sequence of production applications
S ⇒ ... ⇒ ... is a derivation
Can write derivation in linear form or tree form
E → E * E | E + E | ( E ) | id

Leftmost derivation
E ⇒ E + E ⇒ E * E + E ⇒ id * E + E ⇒ id * id + E ⇒ ...

Rightmost derivation
E ⇒ E * E ⇒ E + E ⇒ E * E + id ⇒ E * id + id ⇒ id * id + id

Parse trees
Can use parse tree -
- describes hierarchy
- filters out the order of replacement
Parse tree
- internal nodes are non-terminals
- leaves are terminal
Consider the string
id * id + id * id
Can draw 3 different trees!

Correct tree though is:
When more than one tree derives the same string,

- the grammar is ambiguous -
- need unambiguous grammars

Ambiguity is the property of a grammar and not the language - rewrite the grammar

Problem: precedence & associatively not specified

Consider precedence first

\[
E \rightarrow E \ast T \mid E + T
\]

string id + id * id + id

\[
\begin{array}{c}
\text{E } + \text{T} \\
\text{E } \ast \text{T} \\
\text{E } + \text{T} \\
\text{T} \\
\text{id}
\end{array}
\]

\[
\begin{array}{c}
\text{E } \ast \text{E} \\
\text{E } + \text{E} \\
\text{T} \\
\text{id}
\end{array}
\]

Two different orders of evaluation - neither correct

Soln: build precedence into grammar have different nonterminal for each precedence level

- lowest level - highest in tree (lowest precedence)
- highest level - lower in tree
- same level - same precedence

\[
E \rightarrow E \ast T \mid E - T \mid T
\]

\[
T \rightarrow E \ast F \mid T/F \mid F
\]

\[
F \rightarrow P \mid P \ast F
\]

\[
P \rightarrow \text{var} \mid \text{constant} \mid ( E )
\]
Another problem is associativity.
Recursion on both sides - associativity is unclear.

\[ E \rightarrow E + E \quad E \rightarrow E + E \]

Left recursion - left associate
Right recursion - right associate

\[ E \rightarrow E \cdot T \quad \text{left associate} \]
\[ E \rightarrow F ^ E \quad \text{right associate} \]

Grammars - discussed from point of view of generation
Syntax analysis - processing a string already constructed
- Is string in the language? - recognition
- If yes - compose derivation

Properties of BNF (important for syntax analysis)
- Any BNF grammar has a decidable parsing program
  - Can decide whether or not a string is in language.
- It is undecidable in general whether an arbitrary BNF grammar is
  - unambiguous.
- It is undecidable whether 2 grammars generate same language

Parsing
- Check if string is in language
- Construct parse tree or some representation

Types of Parsers

1. Universal Parsing - can parse any BNF grammar
   - e.g., Early's algorithm - powerful but too inefficient

2. Top-down - goal directed - expands string
   - Only works for certain class of grammars
   - Starts at root of parse tree and tries to get to leaves
   - Leftmost derivation
   - Can be efficiently written by hand
Before we explore parsing techniques, look ahead into what parser produces.

### Abstract Syntax Trees

- A parser traces the derivation of a sequence of tokens
- The rest of the compiler needs a structural representation of the program
- Abstract syntax trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

---

**Abstract Syntax Tree. (Cont.)**

- Consider the grammar
  
  \[
  E \rightarrow \text{int} | ( E ) | E + E
  \]
- And the string
  
  \[
  5 + (2 + 3)
  \]
- After lexical analysis (a list of tokens)
  
  \[
  \text{int}_5 \cdot \text{\texttt{+}} \cdot ( \text{int}_2 \cdot \text{\texttt{+}} \cdot \text{int}_3 )
  \]
- During parsing we build a parse tree ...
Example of Parse Tree

- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

Abstract Syntax Tree

- Also captures the nesting structure
- But abstracts from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler

Semantic Actions

- This is what we’ll use to construct ASTs
- Each grammar symbol may have attributes
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an action
  - Written as:  \( X \rightarrow Y_1 \ldots Y_n \{ \text{action} \} \)
  - That can refer to or compute symbol attributes
Semantic Actions: An Example

- Consider the grammar
  \[ E \rightarrow \text{int} \mid E + E \mid (E) \]
- For each symbol X define an attribute X.val
  - For terminals, val is the associated lexeme
  - For non-terminals, val is the expression's value (and is computed from values of subexpressions)
- We annotate the grammar with actions:
  \[
  \begin{align*}
  E &\rightarrow \text{int} \quad \{ E.val = \text{int}.val \} \\
  | & E_1 + E_2 \quad \{ E.val = E_1.val + E_2.val \} \\
  | & ( E_1 ) \quad \{ E.val = E_1.val \}
  \end{align*}
  \]

Semantic Actions: An Example (Cont.)

- String: 5 + (2 + 3)
- Tokens: int5 '+' '(' int2 '+' int3 ')

<table>
<thead>
<tr>
<th>Productions</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E_1 + E_2</td>
<td>E.val = E_1.val + E_2.val</td>
</tr>
<tr>
<td>E_1 \rightarrow \text{int}_5</td>
<td>E_1.val = \text{int}_5.val = 5</td>
</tr>
<tr>
<td>E_2 \rightarrow ( E_3 )</td>
<td>E_2.val = E_3.val</td>
</tr>
<tr>
<td>E_3 \rightarrow E_4 + E_5</td>
<td>E_3.val = E_4.val + E_5.val</td>
</tr>
<tr>
<td>E_4 \rightarrow \text{int}_2</td>
<td>E_4.val = \text{int}_2.val = 2</td>
</tr>
<tr>
<td>E_5 \rightarrow \text{int}_3</td>
<td>E_5.val = \text{int}_3.val = 3</td>
</tr>
</tbody>
</table>

Semantic Actions: Notes

- Semantic actions specify a system of equations
- Order of resolution is not specified
- Example:
  \[ E_3.val = E_4.val + E_5.val \]
  - Must compute E_4.val and E_5.val before E_3.val
  - We say that E_3.val depends on E_4.val and E_5.val
  - The parser must find the order of evaluation
Lecture 5 CS 2210

**Dependency Graph**

- Each node labeled E has one slot for the val attribute
- Note the dependencies

![Dependency Graph Diagram]

Lecture 5 CS 2210

**Evaluating Attributes**

- An attribute must be computed after all its successors in the dependency graph have been computed
  - In previous example attributes can be computed bottom-up
- Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal

Lecture 5 CS 2210

**Semantic Actions: Notes (Cont.)**

- **Synthesized** attributes
  - Calculated from attributes of descendents in the parse tree
  - E.val is a synthesized attribute
  - Can always be calculated in a bottom-up order
- Grammars with only synthesized attributes are called S-attributed grammars
  - Most frequent kinds of grammars
Semantic Actions: Notes (Cont.)

- Semantic actions can be used to build ASTs
- And many other things as well
  - Also used for type checking, code generation, ...
- Process is called syntax-directed translation
  - Substantial generalization over CFGs

Constructing An AST

- We first define the AST data type
  - Supplied by us for the project
- Consider an abstract tree type with two constructors:
  
  \[
  \text{mkleaf}(n) = \begin{array}{c}
  n \\
  \end{array}
  \]

  \[
  \text{mkplus}(1, 2) = \begin{array}{c}
  \text{PLUS} \\
  1 \quad 2
  \end{array}
  \]

Constructing a Parse Tree

- We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int_lexval is the value of the integer lexeme
  - Computed using semantic actions

\[
E \rightarrow \text{int} \quad E.ast = \text{mkleaf}(\text{int.lexval})
\]

\[
| E_1 \ast E_2 \quad E.ast = \text{mkplus}(E_1.ast, E_2.ast)
\]

\[
| (E) \quad E.ast = E.ast
\]
### Parse Tree Example

- Consider the string `int 5 '+' '(' int2 '+' int3 ')'
- A bottom-up evaluation of the ast attribute:
  
  \[
  \text{E.ast} = \text{mkplus}(\text{mkleaf}(5), \text{mkplus}(\text{mkleaf}(2), \text{mkleaf}(3)))
  \]

### Review

- We can specify language syntax using CFG
- A parser will answer whether \( s \in \text{L}(G) \)
- ... and will build a parse tree
- ... which we convert to an AST
- ... and pass on to the rest of the compiler
- This and other lectures
  - How do we answer \( s \in \text{L}(G) \) and build a parse tree?
  - After that: from AST to assembly language

### Parsing

- Two approaches
  - Top down
    - Easier to understand and implement manually
  - Bottom up
    - More powerful - can be implemented automatically
Example Grammar

\[ \text{<program>} \rightarrow \text{begin} \text{<stmts>} \text{end} \$ \]
\[ \text{<stmts>} \rightarrow \text{<stmt>} ; \text{<stmts>} \]
\[ \text{<stmts>} \rightarrow \epsilon \]
\[ \text{<stmt>} \rightarrow \text{simplestms} | \text{begin} \text{<stmts>} \text{end} \]

Consider Grammar:

\[ S \rightarrow AB \quad B \rightarrow bD \]
\[ A \rightarrow aC \quad D \rightarrow d \]
\[ C \rightarrow c \]

Consider string derivation:

\[ S \Rightarrow \text{a c b d} \]

\[ S \Rightarrow AB \quad (1) \]
\[ S \Rightarrow AB \quad (5) \]
\[ S \Rightarrow aC B \quad (2) \]
\[ S \Rightarrow A b D \quad (4) \]
\[ S \Rightarrow a C B d \quad (3) \]
\[ S \Rightarrow A b d \quad (3) \]
\[ S \Rightarrow a C b d \quad (4) \]
\[ S \Rightarrow a C b d \quad (2) \]
\[ S \Rightarrow a C b d \quad (2) \]

Leftmost

\[ S \rightarrow A B \quad (1) \]
\[ S \rightarrow A B \quad (5) \]
\[ S \rightarrow a C b D \quad (2) \]
\[ S \rightarrow a C b d \quad (4) \]

Top down & bottom up derivations

Top Down Parsers

1. Recursive descent - simple to implement, uses backtracking
2. Predictive parser - predict the rule based on the 1st m symbols without backtracking - restrictions on the grammar to avoid backtracking
3. LL(k) - predictive parser for LL(k) grammar
   - non recursive & only k symbol look ahead
   - table driven - efficient
Recursive Descent

\[
E \rightarrow T + E \mid T \\
T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
\]

Input string: int*int

Initial parse tree is E

Initial focus is at beginning of output

1. pick a production for E: \( E \rightarrow T \)
2. pick a production for T: \( T \rightarrow (E) \) match current input token int with ( - failure, backtrack to T
3. Pick another production - \( T \rightarrow \text{int} \ast T \) move to int match current input; success; advance input, move to \( \ast \), match current input; success! advance input, move to T
4. Pick production for T: \( T \rightarrow \text{int} \) move to int, match current input success! input end, complete derivation

ACCEPT

Recursive descent parsing uses backtracking

- For a non-terminal in the derivation, productions are tried in some order until
  - a production is found that generates a portion of the input or
  - no production is found that generates a portion of the input, in which case backtrack to previous non-terminal
- Parsing fails if no production for the start symbol generates the entire input
- Terminals of the derivation are compared against input
  - match ~ advance input, continue
  - no match ~ backtrack

Implementation

Create a procedure for each non-terminal which does the following

1. checks if input symbol matches a terminal symbol in the grammar rule
2. calls other procedure when non-terminals are part of the rule
3. if end of procedure is reached, success is reported to the caller
**Problem:** doesn’t work if grammar is left recursive

Right recursion is ok – string is consumed

Left recursion – infinite loop – not consuming string

\[ A \rightarrow A \ a \ | \ B \]

Left recursion can always be removed

Rewrite the grammar so productions “make” some progress

---

**Changing a grammar to remove left recursion**

Example: \( A \rightarrow AB \mid C \)

can be rewritten

- \( A \rightarrow CA' \)
- \( A' \rightarrow BA' \mid \epsilon \)

Note that the item \( C \), which is the first element of every string in the language is generated first
In general can eliminate all immediate left recursion:

\[ A \rightarrow A \alpha \mid \beta \] then change to

\[ A \rightarrow \beta A' \]
\[ A \rightarrow \alpha A' \mid \epsilon \]

Example: \[ <\text{expr}> \rightarrow <\text{expr}> + <\text{term}> \mid <\text{term}> \]
change to

\[ <\text{expr}> \rightarrow <\text{term}> <\text{expr}'> \]
\[ <\text{expr}'> \rightarrow + <\text{term}> <\text{expr}'> \mid \epsilon \]

Not all left recursion is immediate

May be hidden by multiple productions

\[ A \rightarrow BC \mid D \]
\[ B \rightarrow A E \mid F \]

See text section 4.3 for elimination of general left recursion

---

Summary of recursive descent

Recursive descent is a simple and general parsing strategy

- Left-recursion must be eliminated first—can be eliminated automatically
- Unpopular because it is thought to be too inefficient—
  Backtracking reparses the string
  Also must undo semantic actions!
- Techniques used in practice do no backtracking at
  the cost of restricting the class of grammar