PARSING

Baishakhi Ray

Fall 2018

These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)
Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages

- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications

- Many Languages are not regular
Automata that accepts odd numbers of 1

How many 1s it has accepted?
- Only solution is duplicate state

Automata does not have any memory
Intro to Parsing

- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications

- Consider the language \{ (i^i) \mid i \geq 0 \}
  - (), ((())), (((())))
  - ((1 + 2) * 3)

- Nesting structures
  - if.. if.. else.. else..
Intro to Parsing

- Input: if(x==y) 1 else 2;

- Parser Input (Lexical Input):
  
  KEY(IF) ‘(‘ ID(x) OP(‘==’) ‘)’ INT(1) KEY(ELSE) INT(2) ‘;’

- Parser Output

```
  IF-THEN-ELSE
      ==
      =
      =
      ID
      ID
      INT
      INT
```
Nor every string of tokens are valid

Parser must distinguish between valid and invalid token strings.

We need

- A Language: to describe valid string
- A method: to distinguish valid from invalid.
Context Free Grammar

- A CFG consists of
  - A set of terminal $T$
  - A set of non-terminal $N$
  - A start symbol $S$ ($S \in N$)
  - A set of production rules
    - $X \rightarrow Y_1 \ldots Y_N$
    - $X \in N$
    - $Y_i \in \{N, T, \varepsilon\}$

- Ex: $S \rightarrow (S) | \varepsilon$
  - $N = \{S\}$
  - $T = \{( , ) , \varepsilon\}$
Context Free Grammar

1. Begin with a string with only the start symbol S
2. Replace a non-terminal X with in the string by the RHS of some production rule: X→Y₁.....Yₙ
3. Repeat 2 again and again until there are no non-terminals

\[ X₁.....Xᵢ \underbrace{X}^{Xᵢ₊₁} \underbrace{... Xₙ \rightarrow X₁.....Xᵢ \underbrace{Y₁.....Yₖ Xᵢ₊₁}^{... Xₙ}} \]

For the production rule X → Y₁.....Yₖ

\[ \alpha_0 \rightarrow \alpha_1 \rightarrow ... \rightarrow \alpha_n \]

\[ \alpha_0 \rightarrow^{*} \alpha_n, \ n \geq 0 \]
Let $G$ be a CFG with start symbol $S$. Then the language $L(G)$ of $G$ is:

$$\{a_1 \ldots \ldots an | \forall i \ a_i \in T \land S \rightarrow a_1a_2 \ldots \ldots an\}$$
There are no rules to replace terminals.

Once generated, terminals are permanent

Terminals ought to be tokens of programming languages

Context-free grammars are a natural notation for this recursive structure
CFG: Simple Arithmetic expression

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

Languages can be generated: id, ( id ), ( id + id ) \ast id, ...

Derivation

A derivation is a sequence of production
- S $\rightarrow$ ... $\rightarrow$ ...

A derivation can be drawn as a tree
- Start symbol is tree’s root
- For a production $X \rightarrow Y_1...Y_n$, add children $Y_1...Y_n$ to node $X$
- Grammar
  - $E \rightarrow E + E \mid E * E \mid (E) \mid id$

- String
  - $id * id + id$

- Derivation
  - $E \rightarrow E + E$
    - $\rightarrow E * E + E$
    - $\rightarrow id * E + E$
    - $\rightarrow id * id + E$
    - $\rightarrow id * id + id$
Parse Tree

- 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
Parse Tree

- Left-most derivation
  - At each step, replace the left-most non-terminal

E -> E + E
   -> E * E + E
   -> id * E + E
   -> id * id + E
   -> id * id + id

- Right-most derivation
  - At each step, replace the right-most non-terminal

E -> E + E
   -> E + id
   -> E * id + id
   -> id * id + id

Note that, right-most and left-most derivations have the same parse tree
Ambiguity

- **Grammar**
  - $E \rightarrow E + E \mid E \ast E \mid (E) \mid id$

- **String**
  - $id \ast id + id$

---

- **Diagram**

  ![Diagram](image_url)
Ambiguity

- A grammar is ambiguous if it has more than one parse tree for a string
  - There are more than one right-most or left-most derivation for some string

- Ambiguity is bad
  - Leaves meaning for some programs ill-defined
Error Handling

- **Purpose of the compiler is**
  - To detect non-valid programs
  - To translate the valid ones

- **Many kinds of possible errors (e.g., in C)**

<table>
<thead>
<tr>
<th>Error Kind</th>
<th>Example</th>
<th>Detected by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical</td>
<td>... $ ...</td>
<td>Lexer</td>
</tr>
<tr>
<td>Syntax</td>
<td>... x*%...</td>
<td>Parser</td>
</tr>
<tr>
<td>Semantic</td>
<td>... int x; y = x(3);...</td>
<td>Type Checker</td>
</tr>
<tr>
<td>Correctness</td>
<td>your program</td>
<td>tester/user</td>
</tr>
</tbody>
</table>
Error Handling

- **Error Handler should**
  - Recover errors accurately and quickly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- **Types of Error Handling**
  - Panic mode
  - Error productions
  - Automatic local or global correction
Panic Mode Error Handling

- Panic mode is simplest and most popular method

- When an error is detected
  - Discard tokens until one with a clear role is found
  - Continue from there

- Typically looks for “synchronizing” tokens
  - Typically the statement of expression terminators
Panic Mode Error Handling

- Example:
  - $(1 + + 2) + 3$

- Panic-mode recovery:
  - Skip ahead to the next integer and then continue

- Bison: use the special terminal `error` to describe how much input to skip
  - $E \rightarrow \text{int} \mid E + E \mid (E) \mid \text{error}\ \text{int} \mid (\text{error})$

---

Normal mode     Error mode
Error Productions

- Specify known common mistakes in the grammar

- Example:
  - Write $5x$ instead of $5 \times x$
  - Add production rule $E \rightarrow .. \mid E \ E$

- Disadvantages
  - complicates the grammar
Error Corrections

- **Idea: find a correct “nearby” program**
  - Try token insertions and deletions (goal: minimize edit distance)
  - Exhaustive search

- **Disadvantages**
  - Hard to implement
  - Slows down parsing of correct programs
  - “Nearby” is not necessarily “the intended” program
Error Corrections

- **Past**
  - Slow recompilation cycle (even once a day)
  - Find as many errors in once cycle as possible

- **Present**
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling
Abstract Syntax Trees

- A parser traces the derivation of a sequence of tokens

- But 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 Trees

- **Grammar**
  - \[ E \rightarrow \text{int} \mid (\ E \ ) \mid E + E \]

- **String**
  - \[ 5 + (2 + 3) \]

- **After lexical analysis**
  - \[ \text{Int<5>} \ ' + ' (\ ' \text{Int<2>} \ ' + ' \text{Int<3>} \ ' ) \]
Abstract Syntax Trees: $5 + (2 + 3)$

Parse Trees

```
E
  /   \
E +   E
  |    |
Int<5> ( E )
  |    |    |
Int<2> +   E
      |    |
Int<3>
```
Abstract Syntax Trees: $5 + (2 + 3)$

- Parse Trees

- Have too much information
  - Parentheses
  - Single-successor nodes
Abstract Syntax Trees: \(5 + (2 + 3)\)

- Parse Trees
  - ASTs capture the nesting structure
  - But abstracts from the concrete syntax
  - More compact and easier to use

- Have too much information
  - Parentheses
  - Single-successor nodes

- AST
  - \(+\)
  - \(\text{Int}<5>\)
  - \(\text{Int}<2>\)
  - \(\text{Int}<3>\)
Disadvantages of ASTs

- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch

- Expressions in AST may be complex, nested
  - \((x \times y) + (z > 5 \ ? 12 \times z : z + 20)\)

- Want simpler representation for analysis
  - ...at least, for dataflow analysis