What is Parsing?

- Given a grammar and a token string:
  - determine if the grammar can generate the token string?
  - i.e., is the string a legal program in the language?
- In other words, to construct a parse tree for the token string.

What's significant about parse tree?

A parse tree gives a unique syntactic structure

- Leftmost, rightmost derivation
- There is only one leftmost derivation for a parse tree, and symmetrically only one rightmost derivation for a parse tree.

What's significant about parse tree?

A parse tree has a unique meaning, thus provides basis for semantic analysis.

(Syntax-directed semantics: semantics are attached to syntactic structure.)

Relationship among language, grammar, parser

- Chomsky Hierarchy
  http://en.wikipedia.org/wiki/Chomsky_hierarchy
- A language can be described by multiple grammars, while a grammar defines one language.
- A grammar can be parsed by multiple parsers, while a parser accepts one grammar, thus one language.
- Should design a language that allows simple grammar and efficient parser
- For a language, we should construct a grammar that allows fast parsing
- Given a grammar, we should build an efficient parser
Ambiguity

- **Ambiguous grammar:** There can be multiple parse trees for the same sentence (program)
  - In other words, multiple leftmost derivations.

- **Why is it bad?**
  - Multiple meanings

---

Deal with Ambiguity

- **Unambiguous Grammar**
  - Rewrite the grammar to avoid ambiguity.

---

Eliminating Ambiguity for Precedence

- Establish "precedence cascade": using different structured names for different constructs, adding grammar rules.
  - Higher precedence : lower in cascade

  $$\text{expr} \rightarrow \text{expr} + \text{expr} | \text{expr} * \text{expr} | (\text{expr}) | \text{number}$$

  $$\text{expr} \rightarrow \text{expr} + \text{expr} | \text{term}$$

  $$\text{term} \rightarrow \text{term} * \text{term} | (\text{expr}) | \text{number}$$

---

Example of Ambiguity: Precedence

Two different parse trees for expression $3+4*5$

---

Example of Ambiguity: Associativity

Two different parse trees for expression $5-2-1$

- $-$ is right-associative, which is against common practice in integer arithmetic
- $-$ is left-associative, which is what we usually assume
Associativity

- **Left-Associative:** `+`, `-`, `*`, `/`
- **Right-Associative:** `=`

What is meant by `a=b=c=1`?

Eliminating Ambiguity for Associativity

- **Left-associativity:** left-recursion
  
  \[
  \text{expr} \rightarrow \text{expr} - \text{term} \mid \text{term}
  \]

- **Right-associativity:** right-recursion
  
  \[
  \text{expr} \rightarrow \text{expr} = \text{expr} \mid a \mid b \mid c
  \]

Putting Together

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} - \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} / \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow (\text{expr}) \mid \text{number} \\
\text{number} & \rightarrow \text{digit} \mid \text{number} \text{digit}
\end{align*}
\]

We want to make `-` left-associative and `/` has precedence over `-`.

Example of Ambiguity: Dangling-Else

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if ( expr ) stmt} \\
\text{if-stmt} & \rightarrow \text{if ( expr ) stmt else stmt} \\
\text{if-stmt} & \rightarrow \text{if ( expr ) stmt [ else stmt ]}
\end{align*}
\]

Two different parse trees for "if(expr) if(expr) stmt else stmt"

Eliminating Dangling-Else

\[
\begin{align*}
\text{stmt} & \rightarrow \text{matched_stmt} \\
\text{unmatched_stmt} & \rightarrow \text{if ( expr ) matched_stmt else matched_stmt} \\
\text{matched_stmt} & \rightarrow \text{if ( expr ) stmt} \\
\text{unmatched_stmt} & \rightarrow \text{if ( expr ) stmt}
\end{align*}
\]

EBNF

- **Repetition { }**
  
  \[
  \text{number} \rightarrow \text{digit} \mid \text{number digit} \\
  \text{expr} \rightarrow \text{expr} - \text{term} \mid \text{term}
  \]

- **Option [ ]**
  
  \[
  \text{signed-number} \rightarrow \text{sign number} \mid \text{number} \\
  \text{if-stmt} \rightarrow \text{if ( expr ) stmt} \\
  \text{if-stmt} \rightarrow \text{if ( expr ) stmt else stmt}
  \]

© Chengkai Li, Weimin He, Weimin He 2008
Syntax Diagrams

- Written from EBNF, not BNF
- If-statement
  (more examples on page 101)

```
if
  expression
else
  statement
```

Parsing Techniques

- Intuitive analysis and conclusion
- No formal theorems and rigorous proofs
- More details: compilers, automata theory

Parsing

- Parsing:
  - Determine if a grammar can generate a given token string.
  - That is, to construct a parse tree for the token string.
- Two ways of constructing the parse tree
  - Top-down (from root towards leaves)
    - Can be constructed more easily by hand
    - Bottom-up (from leaves towards root)
      - Can handle a larger class of grammars
      - Parser generators tend to use bottom-up methods

Top-Down Parser

- Recursive-descent parser:
  - A special kind of top-down parser: single left-to-right scan, with one lookahead symbol.
  - Backtracking (trial-and-error) may happen
- Predictive parser:
  - The lookahead symbol determines which production to apply, without backtracking

Recursive-Descent Parser

- Types in Pascal
  type → simple | array | simple { of type }
  simple → char | integer

```
Input:
array [ integer ] of char
Parse tree
```

Challenge 1: Top-Down Parser Cannot Handle Left-Recursion

```
expr → expr - term | term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
Input:
3 - 4 - 5
Parse tree
```

```
expr
  expr - term

expr
  term
```

```
```
Eliminating Left-Recursion

expr → expr - term | term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

expr → term expr’ | term expr’ | ε
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(expr)

void expr(void)
{ term();
while (token == '-')
{ match('-');
term();
}
}

Challenge 2: Backtracking is Inefficient

subscription → term | term .. term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

• We cannot avoid backtracking if the grammar has multiple productions to apply, given a lookahead symbol.

• Solution:
  - Change the grammar so that there is only one applicable production that is unambiguously determined by lookahead symbol.

Avoiding Backtracking by Left Factoring

A → α β1 | α β2
A’ → α A’

expr → term | term .. term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

expr → term rest
rest → term |
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Left Factoring Using EBNF

expr → @ expr | term
expr → @ expr
if-statement → if ( expr ) statement
  | if ( expr ) statement else statement

if-statement → if ( expr ) statement [ else statement ]