The Structure of a Compiler

- Program text
- Syntactic analyzer
- Intermediate representation
- Code generator
- Machine code
- Tokenizer
- Token stream
- Parser
Role of the Parser

- Converts a token stream into an intermediate representation
  - Captures the meaning (instead of text) of the program
  - Usually, intermediate representation is a parse tree

\[
x = x + 1
\]

\[
<\text{id, "x"}> <\text{assign}> <\text{id, "x"}> <\text{plus}> <\text{number, "1"}>
\]
Kinds of Parsers

• Universal
  – Can parse any grammar
  – Cocke-Younger-Kasami and Earley's algorithms
  – Not efficient enough to be used in compilers

• Top-down
  – Builds parse trees from the top (root) down

• Bottom-up
  – Builds parse trees from the bottom (leaves) up
Errors in Parsing

- **Lexical errors**
  - Misspelled identifiers, keywords, or operators

- **Syntactical errors**
  - Misplaced or mismatched parentheses, case statement outside of any switch statement,...

- **Semantic errors**
  - Type mismatches between operators and operands

- **Logical errors**
  - Bugs - the programmer said one thing but meant something else
    - `if (x = y) { ... }`
    - `if (x == y) { ... }`
Error Reporting

• A parser should
  – report the presence of errors clearly and correctly
  – recover from errors quickly enough to detect further errors
Error Recovery Modes

• Panic-Mode
  – discard input symbols until a "synchronizing token" is found
  – Examples (in Java): semicolon, '}'

• Phrase-Level
  – replace a prefix of the remaining input to correct it
  – Example: Insert ';' or '{'
  – must be careful to avoid infinite loops
Error Recover Modes (Cont'd)

- Error Productions
  - Specify common errors as part of the language specification

- Global Correction
  - Compute the smallest set of changes that will make the program syntactically correct (impractical and usually not usually useful)
Context-Free Grammars

• CF grammars are used to define languages
• Specified using BNF notation
  – A set of non-terminals $N$
  – A set of terminals $T$
  – A list of rewrite rules (productions)
  – The LHS of each rule contains one non-terminal symbol
  – The RHS of each rule contains a regular expression over the alphabet $N \cup T$
  – A special non-terminal is usually designated as the start symbol
    • Usually, start symbols is LHS of the first production
Context Free Grammars and Compilers

• In a compiler
  – $N$ consists of language constructs (function, block, if-statement, expression, ...)
  – $T$ consists of tokens
**Grammar Example**

- **Non-terminals:** E, T, F
  - E = expression
  - T = term
  - F = factor
- **Terminals:** id, +, *, (, )
- **Start symbol:** E
- "Mathematical formulae using + and *"

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E + E</td>
</tr>
<tr>
<td>E → E + T</td>
</tr>
<tr>
<td>T → T * F</td>
</tr>
<tr>
<td>F → ( E )</td>
</tr>
<tr>
<td>E → T E'</td>
</tr>
<tr>
<td>E' → + T E'</td>
</tr>
<tr>
<td>T → F T'</td>
</tr>
<tr>
<td>T' → * F T'</td>
</tr>
<tr>
<td>F → ( E )</td>
</tr>
</tbody>
</table>
Derivations

• From a grammar specification, we can derive any string in the language
  – Start with the start symbol
  – While the current string contains some non-terminal $N$
    • expand $N$ using a rewrite rule with $N$ on the LHS

$E \rightarrow E + E \mid E \ast E \mid (E) \mid id$


E
E + E
E + id
E \ast E + id
E \ast id + id
(E) \ast id + id
(E + E) \ast id + id
(id + E) \ast id + id
(id + id) \ast id + id
Derivation Example

• Derive id + id * id with these grammars:

\[
E → E + E \mid E * E \mid (E) \mid \text{id}
\]

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

\[
E → T E'
E' → + T E' \mid \epsilon
T → F T'
T' → * F T' \mid \epsilon
T → T * F \mid F
F → (E) \mid \text{id}
\]
Derivation Example

• Derive:
  - \( \text{id} \times \text{id} + \text{id} \)
  - \( \text{id} + \text{id} \times \text{id} \)

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

- The strings of terminals that we derive from the start symbol are called *sentences*
- The strings of terminals and non-terminals are called *sentential forms*
Leftmost and Rightmost Derivations

- A derivation is leftmost if at each stage we always expand the leftmost non-terminal.
- A derivation is rightmost if at each stage we always expand the rightmost non-terminal.
- Give a leftmost and rightmost derivation of
  - id * id + id

E → E + E | E * E | ( E ) | id
Derivations and Parse Trees

• A parse-tree is a graphical representation of a derivation

• Internal nodes are labelled with non-terminals
  – Root is the start symbol

• Leaves are labelled with terminals
  – String is represented by left-to-right traversal of leaves

• When applying an expansion $E \rightarrow ABC\ldots Z$
  – Children of node $E$ become nodes labelled $A, B, C, \ldots Z$
Derivations and Parse Trees - Example

E → E + E | E * E | ( E ) | id

E
E + E
E + id
E * E + id
E * id + id
(E) * id + id
(E + E) * id + id
(id + E) * id + id
(id + id) * id + id
**Ambiguity**

- Different parse trees for the same sentence result in ambiguity

E → E + E | E * E | (E) | id
**Ambiguity - Cont'd**

- Ambiguity is usually bad
- The same program means two different things
- We try to write grammars that avoid ambiguity
Context Free Grammars and Regular Expressions

• CFGs are more powerful than regular expressions
  – Converting a regular expression to a CFG is trivial
  – The CFG $S \rightarrow aSb \mid \varepsilon$ generates a language that is not regular

• But not that powerful
  – The language $\{ a^m b^n c^m d^n : n, m > 0 \}$ can not be expressed by a CFG
Enforcing Order of Operations

- We can write a CFG to enforce specific order of operations
  - Example: + and *
  - Exercises:
    - Add comparison operator with lower level of precedence than +
    - Add exponentiation operator with higher level of precedence than *

E → PE
PE → TE + PE | TE
TE → id * TE | id | ( E )
Picking Up - Context free grammars

• CFGs can specify programming languages
• It's not enough to write a correct CFG
  – An ambiguous CFG can give two different parse trees for the same string
    • Same program has two different meanings!
    – Not all CFGs are easy to parse efficiently
• We look at restricted classes of CFGs
  – Sufficiently restricted grammars can be parsed easily
  – The parser can be generated automatically
Parser Generators

- **Benefits of parser generators**
  - No need to write code (just grammar)
  - Parser always corresponds exactly to the grammar specification
  - Can check for errors or ambiguities in grammars
  - No surprise programs

- **Drawbacks**
  - Need to write a restricted class of grammar [LL(1), LR(1), LR(k),...]
  - Must be able to understand when and why a grammar is not LL(1) or LR(1) or LR(k)
  - Means learning a bit of parsing theory
  - Means learning how to make your grammar LL(1), LR(1), or LR(k)
Ambiguity

• This grammar is ambiguous
  – consider the input
    • \( a - b - c \)
• Rewrite this grammar to be unambiguous
• Rewrite this grammar so that - becomes left associative:
  • \( a - b - c \sim ( (a - b) - c ) \)
Solutions

\[
\begin{align*}
E & \rightarrow \text{id} \ M \\
M & \rightarrow - \ E \mid \varepsilon \\
E & \rightarrow \text{id} \\
M & \rightarrow E - \text{id}
\end{align*}
\]
A Common Ambiguity - The Dangling Else

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
| \text{if expr then stmt else stmt} \\
| \text{other}
\]

• Show that this grammar is ambiguous
• Remove the ambiguity
  – Implement the “\text{else} matches innermost \text{if}” rule
Solution

\[
\begin{align*}
\text{stmt} & \rightarrow \text{matched_stmt} \mid \text{open_stmt} \\
\text{matched_stmt} & \rightarrow \text{if expr then matched_stmt} \\
& \quad \text{else matched_stmt} \\
& \quad \mid \text{other} \\
\text{open_stmt} & \rightarrow \text{if expr then stmt} \\
& \quad \mid \text{if expr then matched_stmt} \\
& \quad \text{else open_stmt}
\end{align*}
\]
Left Recursion

- A top-down parser expands the left-most non-terminal based on the next token
- Left-recursion is difficult for top-down parsing
- Immediate left recursion:
  - $A \rightarrow A\alpha \mid \beta$
  - Rewrite as: $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \epsilon$
- More complicated left recursion occurs when $A$ can derive a string starting with $A$
  - $A \rightarrow^+ A\alpha$
Removing Left Recursion

- Removing immediate left-recursion is easy
- Simple case:
  - \[ A \rightarrow A\alpha \mid \beta \]
  - Rewrite as: \[ A \rightarrow \beta A' \] and \[ A' \rightarrow \alpha A' \mid \epsilon \]
- More complicated:
  - \[ A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_\kappa \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_\tau \]
  - Rewrite as:
    - \[ A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_\tau A' \]
    - \[ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_\kappa A' \mid \epsilon \]
Algorithm for Removing all Left Recursion
• Textbook page 213
Left Factoring

- Left factoring is a technique for making a grammar suitable for top-down parsing.
- For each non-terminal $A$ find the longest prefix $\alpha$ common to two or more alternatives.
  - Replace $A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3 | \ldots | \alpha \beta_n$ with
  - $A \rightarrow \alpha A'$ and $A' \rightarrow \beta_1 | \beta_2 | \beta_3 | \ldots | \beta_n$
- Repeat until no two alternatives have a common prefix.
Left Factoring Example

• Left factor the following grammars

\[
\begin{align*}
E & \rightarrow PE \\
PE & \rightarrow TE + PE \mid TE \\
TE & \rightarrow id \ast TE \mid id \mid (E)
\end{align*}
\]
Summary of Grammar-Manipulation Tricks

• Eliminating ambiguity
  – Different parse trees for same program

• Enforcing order of operations
  – Left-associative
  – Right-associative

• Eliminating left-recursion
  – Gets rid of potential "infinite recursions"

• Left factoring
  – Allows choosing between alternative productions based on current input symbol
Exercise

rexpr → rexpr + rterm | rterm
rterm → rterm rfactor | rfactor
rfactor → rfactor * | rprimary
rprimary → a | b

• Remove left recursion
• Left-factor
Top-Down Parsing

- Top-down parsing is the problem of constructing a pre-order traversal of the parse tree
- This results in a leftmost derivation
- The expansion of the leftmost non-terminal is determined by looking at a prefix of the input
**LL(1) and LL(k)**

- If the correct expansion can always be determined by looking ahead at most k symbols then the grammar is an LL(k) grammar
- LL(1) grammars are most common
**FIRST(α)**

- Let $α$ be any string of grammar symbols
- $FIRST(α)$ is the set of terminals that begin strings that can be derived from $α$
  - If $α$ can derive $ε$ then $ε$ is also in $FIRST(α)$
- **Why is FIRST useful**
  - Suppose $A \rightarrow α \mid β$ and $FIRST(α)$ and $FIRST(β)$ are disjoint
  - Then, by looking at the next symbol we know which production to use next
Computing $FIRST(X)$

- If $X$ is a terminal then $FIRST(X) = \{X\}$
- If $X$ is a non-terminal and $X \rightarrow Y_1 Y_2 ... Y_k$
  - $i = 0$; define $FIRST(Y_0) = \{\varepsilon\}$
  - while $\varepsilon$ is in $FIRST(Y_i)$
    - Add $FIRST(Y_{i+1})$ to $FIRST(X)$
    - $i = i+1$
  - if ($i = k$ or $X \rightarrow \varepsilon$)
    - Add $\varepsilon$ to $FIRST(X)$
- Repeat above step for all non-terminals until nothing is added to any $FIRST$ set
Example

• Compute \( \text{FIRST}(E) \), \( \text{FIRST}(PE) \), \( \text{FIRST}(TE) \), \( \text{FIRST}(TE') \)

\[
\begin{align*}
E & \rightarrow PE \\
PE & \rightarrow TE + E \mid TE \\
TE & \rightarrow id \; TE' \\
TE' & \rightarrow * \; E \\
TE' & \rightarrow id \\
TE' & \rightarrow ( \; E \; )
\end{align*}
\]
Computing $\text{FIRST}(X_1 X_2 \ldots X_k)$

- Given $\text{FIRST}(X)$ for every symbol $X$ we can compute $\text{FIRST}(X_1 X_2 \ldots X_k)$ for any string of symbols $X_1 X_2 \ldots X_k$:
  - $i = 0$; define $\text{FIRST}(X_0) = \{ \varepsilon \}$
  - while $\varepsilon$ is in $\text{FIRST}(X_i)$
    - Add $\text{FIRST}(X_{i+1})$ to $\text{FIRST}(X_1 X_2 \ldots X_k)$
    - $i = i+1$
  - if ($i = k$)
    - Add $\varepsilon$ to $\text{FIRST}(X)$
**FOLLOW(A)**

- Let A be any non-terminal
- FOLLOW(A) is the set of terminals \( a \) that can appear immediately to the right of A in some sentential form
  
  - I.e. \( S \rightarrow^* \alpha A a \beta \) for some \( \alpha \) and \( \beta \) and start symbol \( S \)
  
  - Also, if A can be a rightmost symbol in some sentential form then \( \$ \) (end of input marker) is in FOLLOW(A)
Computing FOLLOW(A)

• Place $ into FOLLOW(S)
• Repeat until nothing changes:
  – if $ \rightarrow \alpha B \beta$ then add FIRST(\beta) \{\epsilon\} to FOLLOW(B)
  – if $ \rightarrow \alpha B$ then add FOLLOW(A) to FOLLOW(B)
  – if $ \rightarrow \alpha B \beta$ and $\epsilon$ is in FIRST(\beta) then add FOLLOW(A) to FOLLOW(B)
Example

• Compute FOLLOW(E), FOLLOW(PE), FOLLOW(TE), FOLLOW(TE')

```
E → PE
PE → TE + E | TE
TE → id TE'
TE' → * E
TE' → id
TE' → ( E )
```
**FIRST and FOLLOW Example**

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

- \( \text{FIRST}(F) = \text{FIRST}(T) = \text{FIRST}(E) = \{ (, \text{id} \} \)
- \( \text{FIRST}(E') = \{ +, \epsilon \} \)
- \( \text{FIRST}(T') = \{ *, \epsilon \} \)
- \( \text{FOLLOW}(E) = \text{FOLLOW}(E') = \{ \), $\} \)
- \( \text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ +, \), $\} \)
- \( \text{FOLLOW}(F) = \{ +, *, \), $\} \)
**LL(1) Grammars**

- Left to right parsers producing a leftmost derivation looking ahead by at most 1 symbol
- Grammar G is LL(1) iff for every two productions of the form $A \rightarrow \alpha \mid \beta$
  - $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ are disjoint
  - If $\varepsilon$ is in $\text{FIRST}(\beta)$ then $\text{FIRST}(\alpha)$ and $\text{FOLLOW}(A)$ are disjoint (and vice versa)
- Most programming language constructs are LL(1) but careful grammar writing is required
LL(1) Predictions Tables

- LL(1) languages can be parsed efficiently through the use of a prediction table
  - Rows are non-terminals
  - Columns are input symbols (terminals)
  - Values are productions
Constructing LL(1) Prediction Table

• The following algorithm constructs the LL(1) prediction table

• For each production $A \rightarrow \alpha$ in the grammar
  – For each terminal $a$ in $\text{FIRST}(\alpha)$, set $M[A,a] = A \rightarrow \alpha$
  – If $\epsilon$ is in $\text{FIRST}(\alpha)$ then for each terminal $b$ in $\text{FOLLOW}(A)$, set $M[A,b] = A \rightarrow \alpha$
### LL(1) Prediction Table Example

<table>
<thead>
<tr>
<th></th>
<th>Id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → T E'</td>
<td></td>
<td></td>
<td>E → T E'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>E' → + T E'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E' → ε</td>
</tr>
<tr>
<td>T</td>
<td>T → F T'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T → F T'</td>
</tr>
<tr>
<td>T'</td>
<td>T' → ε</td>
<td>T' → * F T'</td>
<td></td>
<td></td>
<td>T' → ε</td>
<td>T' → ε</td>
</tr>
<tr>
<td>F</td>
<td>F → id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F → ( E )</td>
</tr>
</tbody>
</table>

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → ( E ) | id
**LL(1) Prediction Table Example**

- Use the table to find the derivation of
  - \( \text{id} + \text{id} \ast \text{id} + \text{id} \)

<table>
<thead>
<tr>
<th></th>
<th>\text{Id}</th>
<th>\ast</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{E}</td>
<td>E \rightarrow T E'</td>
<td></td>
<td>E \rightarrow T E'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{E}'</td>
<td>E' \rightarrow T E'</td>
<td>\rightarrow e</td>
<td>E' \rightarrow \varepsilon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{T}</td>
<td>T \rightarrow F T'</td>
<td></td>
<td>T \rightarrow F T'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{T}'</td>
<td>T' \rightarrow \varepsilon</td>
<td>\rightarrow F T'</td>
<td>T' \rightarrow \varepsilon</td>
<td>T' \rightarrow \varepsilon</td>
<td></td>
</tr>
<tr>
<td>\text{F}</td>
<td>F \rightarrow \text{id}</td>
<td></td>
<td></td>
<td></td>
<td>F \rightarrow (E)</td>
</tr>
</tbody>
</table>
LL(1) Parser Generators

• Given a grammar $G$, an LL(1) parser generator can
  – Compute FIRST($A$) and FOLLOW($A$) for every non-terminal $A$ in $G$
  – Determine if $G$ is LL(1)
  – Construct the prediction table for $G$
  – Create code that parses any string in $G$ and produces the parse tree

• In Assignment 2 we will use such a parser generator (javacc)
Summary

• Programming languages can be specified with context-free grammars
• Some of these grammars are easy to parse and generate a unique parse tree for any program
• An LL(1) grammar is one for which a leftmost derivation can be done with only one symbol of lookahead
• LL(1) parser generators exist and can produce efficient parsers given only the grammar