Agenda

- Recognize, Analyze, Transform
- Lexical analysis
- Building lexical analyzers
Syntax Analysis

• The syntax of a language defines the rules by which a sequence of tokens can be recognized as a legal construction in that language.

• Recognize and distinguish legal and illegal sentences of the language
Addition expressions
Addition expressions tokens as regular expressions

$$\text{Digits} = \text{‘0’ ... ‘9’}$$
$$\text{Plus} = \text{‘+’}$$
$$\text{Literal} = \text{Digits}^+$$

$$\text{Token} = \text{Plus} \mid \text{Literal}$$
Addition expressions as regular expressions

Digits = ( ‘0’ ... ‘9’ )+
Expr = ( Digits ‘+’ )* Digits
Digits = ( '0' … '9' )+
Expr = ( Digits ' +' )* Digits

Expr = ( ( '0' … '9' )+ ' +' )* ( '0' … '9' )+
Expressions with parentheses

Digits = ( ‘0’ ... ‘9’ )+
Add = Expr ‘+’ Expr
Expr = Digits | ‘(’ Add ‘)’

488
(400+88)
(400+(44+44))
((400+4)+(42+42))
Digits = ( ‘0’ … ‘9’ )+
Add = Expr ‘+’ Expr
Expr = Digits | ‘(‘ Add ‘)’

Expr = Digits | ( Expr ‘+’ Expr )

Expr = Digits | ( ( Digits | ( Expr ‘+’ Expr ) ) ’+’ ( Digits | ( Expr ‘+’ Expr ) ) )
Abbreviations in regular expressions are just syntactic sugar...

They add no additional expressive power
Recursion adds expressive power
Recursive abbreviations?

- Simplifies regular expressions
  - Alternation within expressions is no longer required
  - Alternation pipe | is no longer required
  - Kleene closure is no longer required
Recursive abbreviations

\[ \text{prod} = \alpha \beta ( \gamma \mid \delta ) \varepsilon \]

\[ \text{helper} = \gamma \mid \delta \]

\[ \text{prod} = \alpha \beta \text{ helper} \varepsilon \]
Recursive abbreviations

\[
\text{helper} = \gamma | \delta \\
\text{prod} = \alpha \beta \text{ helper} \varepsilon
\]
Recursive abbreviations

\[ \text{prod} = (\alpha \beta \gamma)^* \]

\[ \text{prod} = (\alpha \beta \gamma) \text{ prod} \]
\[ \text{prod} = \lambda \]
This simple but powerful notation of recursive abbreviations is referred to as \textit{context-free grammars (CFG)}.
Context-free grammars

• CFGs can describe richer languages than regular expressions

  • Thus need something more powerful than finite automata to recognize them (key is having memory)

• A language is a set of strings over an alphabet $\Sigma$

• Alphabet $\Sigma$ ranges over tokens, not characters

  • Example: $\Sigma = \{ \text{IF, IDENT, PLUS, LBRACE, EQ, … } \}$

• A CFG consists of a set of productions
Context-free grammars

A production is of the form:

symbol $\rightarrow$ symbol symbol symbol ... symbol

- Right hand side has 0 or more symbols (0 means $\lambda$)
- A symbol is either:
  - *Terminal*: a token from alphabet $\Sigma$
  - *Non-terminal*: appears on the left hand side of a production
- Left hand side symbol is always a non-terminal
- Distinguished *start* production (typically the first)
id = num ;
id = num ;
print id + ( id + num )

x = 400 ;
y = 42 ;
print x + ( y + 46 )
Derivations

- Derive a sentence from the grammar to show that it is in the language

- Begin with the start symbol, and repeatedly replace right hand side non-terminals with symbols from productions

- Many possible derivation orders
  - *Left-most derivation*: always expand the left-most non-terminal first, working towards the right
  - *Right-most derivation*
id = num ;
id = num ;
print id + ( id + num )
Parse trees

For each symbol in the derivation, connect up to its originating symbol
Ambiguous grammars

Two possible parse trees for the same sentence

\[ x = 1 + 2 + 3 \]

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

(1 + 2) + 3

1 + (2 + 3)
Removing ambiguity

We want left-associativity

S → E
E → id
E → num
E → E + E
E → ( E )

S → E
E → T
E → E + T
T → id
T → num
T → ( E )

T for terms because they are added together
Removing ambiguity

S → E
E → T
E → E + T
T → id
T → num
T → ( E )

1 + 2 + 3

S
  |  
  E
   |   +
   T
     |  |
     E
      | |           |
      +  num
        |
        |
        num
More ambiguity

We want multiplication & division to be higher-precedence, or to bind more tightly

S → E
E → id
E → num
E → E * E
E → E / E
E → E + E
E → E - E
E → (E)

S → E
E → T
E → E + T
E → E - T
T → T * F
T → T / F
T → F

F → id
F → num
F → (E)

F for factors because they are multiplied together
Impossible to derive

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow T \\
E & \rightarrow E + T \\
E & \rightarrow E - T \\
T & \rightarrow T \ast F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow \text{id} \\
F & \rightarrow \text{num} \\
F & \rightarrow (E)
\end{align*}
\]

\[(1 + 2) \ast 3\]
Backus-Naur Form (BNF)
Backus-Naur Form (BNF) — ALGOL 60 Report

\[ \begin{align*}
\texttt{<expression>} & \ ::= \texttt{<term>} \\
& \quad \mid \texttt{<expression>} \texttt{"+"} \texttt{<term>} \\
& \quad \mid \texttt{<expression>} \texttt{"-"} \texttt{<term>} \\
\texttt{<term>} & \ ::= \texttt{<factor>} \\
& \quad \mid \texttt{<term>} \texttt{"*"} \texttt{<factor>} \\
& \quad \mid \texttt{<term>} \texttt{"/"} \texttt{<factor>} \\
\texttt{<factor>} & \ ::= \texttt{<identifier>} \\
& \quad \mid \texttt{<number>} \\
& \quad \mid \left( \texttt{<expression>} \right) \\
\end{align*} \]
PLY (Python Lex-Yacc)

expression : term
| expression PLUS term
| expression MINUS term

term : factor
| term STAR factor
| term SLASH factor

factor : identifier
| number
| LPAREN expression RPAREN
Parsing Top Down vs Bottom Up
Top Down Parsing

• Starting from the start symbol S, find the correct sequence of production expansions that will transform S into the input token stream (if possible)

  • Called a derivation

• Build the parse tree from the root (S) to the leaves (terminals)

• Top down techniques: LL(k), Recursive Descent
Bottom Up Parsing

• Given a stream of input tokens, find the correct sequence of production contractions that take the input back to the start symbol

  • Called a *reduction* or *reverse derivation*

• Build the parse tree from the leaves (terminals) to the root (S)

• Bottom up techniques: LR(k), LALR(k), SLR(k)
LL(1) Grammars

Peter McCormick

January 24, 2018
Define a context-free grammar $G = (N, \Sigma, S, P)$ where

- $\Sigma$ is the set of terminals (the alphabet), each represented as lower case Roman letters ($t, x, y, z$)
- $N$ is the set of non-terminals, each represented by capitalized Roman letters ($A, B, X, Y$)
- $S \in N$ is the distinguished start symbol
- $P$ is a finite set of productions
- $V = \Sigma \cup N$ is the vocabulary of the grammar. Strings ranging over $V$, such as $AwXt$, are each represented by Greek letters ($\alpha, \beta, \gamma$)

$X \Rightarrow^+ \alpha$ means that there is a valid sequence of derivations starting from the non-terminal $X$ to the string $\alpha$

Let $\$ be a special end-of-input marker
A production is a *rewriting rule* of the form:

\[ X \rightarrow \alpha_1 \alpha_2 \cdots \alpha_m \]

where \( X \in N \), \( \alpha_i \in V^* \) for each \( i \), and \( m \geq 0 \). If \( m = 0 \), then

\[ X \rightarrow \lambda \]

That is, \( X \) can be replaced by \( \lambda \), the empty string.

- *Rewriting* is the act of replacing a non-terminal \( X \) with the right hand of a production for \( X \), so \( \alpha_1 \cdots \alpha_m \) replacing \( X \)
- \( X \Rightarrow^+ \alpha \) means that there is a valid sequence of derivations starting from the non-terminal \( X \) to the vocabulary string \( \alpha \)
Definitions

A vocabulary string is in *sentential form* if it is in the set of all strings that can be derived from the start symbol:

\[ \{ w : S \Rightarrow^* w \} \]

A *language* is the set of all terminal strings that can be derived from the start symbol:

\[ \{ w : S \Rightarrow^* w \} \cap \Sigma^* \]
Definitions

Given a production \( X \rightarrow \gamma \) and a sentential form \( \alpha X \beta \), then

\[
\begin{align*}
\alpha X \beta \quad &\Rightarrow \quad \alpha \gamma \beta \text{ is a derivation in one step} \\
\alpha X \beta \quad &\Rightarrow^* \quad \alpha \gamma \beta \text{ is a derivation in zero or more steps} \\
\alpha X \beta \quad &\Rightarrow^+ \quad \alpha \gamma \beta \text{ is a derivation in one or more steps}
\end{align*}
\]

- A *left-most derivation* expands non-terminals left to right, while a *right-most derivation* expands right to left
**LL(1) Grammars**

- *LL* is a set of all languages that can be parsed by an *LL* parser.
- An *LL* parser consumes its input **Left-to-right**, producing a **Leftmost-derivation**.
- *LL(k)* means *LL* with *k* tokens worth of input look ahead.
- *LL(1)* means 1 input token lookahead.
Nullable

\[ \alpha \in V^+ \text{ is } \textit{nullable} \text{ iff } \alpha \Rightarrow^* \lambda \]
Nullable

Given the productions:

\[
\begin{align*}
X & \rightarrow YZW \\
Y & \rightarrow \lambda \\
Z & \rightarrow z | Y \\
W & \rightarrow w | YZ \\
\end{align*}
\]

Then \( X \) is nullable since:

\[
\begin{align*}
X & \Rightarrow YZW \\
& \Rightarrow ZW \text{ (since } Y \text{ is nullable)} \\
& \Rightarrow YW \\
& \Rightarrow W \text{ (again since } Y \text{ is nullable)} \\
& \Rightarrow YZ \\
& \Rightarrow Z \text{ (again)} \\
& \Rightarrow Y \Rightarrow \lambda \\
\end{align*}
\]
### First Sets

<table>
<thead>
<tr>
<th>First Set Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$First(z \beta)$</td>
<td>${z}$, $z$ is a terminal symbol</td>
</tr>
<tr>
<td>$First(\lambda)$</td>
<td>${}$, (the empty set)</td>
</tr>
<tr>
<td>$First(B\gamma)$</td>
<td>$First(B)$, (if $B$ is not nullable)</td>
</tr>
<tr>
<td>$= First(B) \cup First(\gamma)$</td>
<td>(if $B$ is nullable)</td>
</tr>
</tbody>
</table>
Follow Sets

$Follow(X) = \{ t \in \Sigma : S \Rightarrow^+ \alpha X t \beta \}$
## Follow Set Construction Rules

Starting from initially empty follow sets, iteratively apply these rules until the sets no longer change.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1. If $S$ is the goal symbol</td>
<td>Add {$} to $\text{Follow}(S)$</td>
</tr>
<tr>
<td>#2. If $S \Rightarrow^+ \alpha X$</td>
<td>Add {$} to $\text{Follow}(X)$</td>
</tr>
<tr>
<td>#3. If $S \Rightarrow^+ \alpha X t \beta$</td>
<td>Add {t} to $\text{Follow}(X)$</td>
</tr>
</tbody>
</table>
| #4. If $S \Rightarrow^+ \alpha X Y \beta$ | If $Y$ is not nullable:  
  · Add $\text{First}(Y)$ to $\text{Follow}(X)$  
Else if $Y$ is nullable:  
  · Add $\text{First}(Y) \cup \text{First}(\beta)$ to $\text{Follow}(X)$ |
| #5. If $X \rightarrow \alpha Y$ | Add $\text{Follow}(X)$ to $\text{Follow}(Y)$ |
Predict Sets for $LL(1)$ Grammars

Given a non-terminal $X$ defined with several alternate productions:

\[
X \rightarrow \gamma_1 \\
\rightarrow \gamma_2 \\
\ldots \\
\rightarrow \gamma_m
\]

The predict set for each production $X \rightarrow \gamma_i$ is defined as

\[
\text{Predict}(X \rightarrow \gamma_i) = \begin{cases} 
\text{First}(\gamma_i) & \text{if } \gamma_i \text{ is not nullable} \\
\text{First}(\gamma_i) \cup \text{Follow}(X) & \text{if } \gamma_i \text{ is nullable}
\end{cases}
\]
Predict Sets for $LL(1)$ Grammars

To be an $LL(1)$ grammar, the Predict sets for all productions for a given non-terminal $X$ must be mutually disjoint.

\[
X \rightarrow Y\alpha \\
\rightarrow Y\beta
\]

is not $LL(1)$ because the Predict sets for the productions of $X$ are not mutually disjoint:

\[
Predict(X \rightarrow Y\alpha) = First(Y\alpha) \text{ (if } Y \text{ is not nullable)} \\
= First(Y) \\
= First(Y\beta) \\
= Predict(X \rightarrow Y\beta)
\]

\[
Predict(X \rightarrow Y\alpha) = First(Y\alpha) \cup \text{Follow}(X) \text{ (if } Y \text{ is nullable)}
\]

\[
Predict(X \rightarrow Y\beta) = First(Y\beta) \cup \text{Follow}(X) \text{ (if } Y \text{ is nullable)}
\]
Factoring Out Common Prefix

Transform:

\[ X \rightarrow Y \alpha \]
\[ \rightarrow Y \beta \]

into:

\[ X \rightarrow Y X_{\text{tail}} \]
\[ X_{\text{tail}} \rightarrow \alpha \]
\[ \rightarrow \beta \]
**Left Recursion**

*LL*(1) grammars cannot handle left recursion:

\[
X \rightarrow X \alpha \\
\rightarrow \beta
\]

If \( \beta \) is *not* nullable:

\[
\text{Predict} (X \rightarrow X \alpha) = \text{First} (X \alpha) \\
= \text{First} (X) \\
= \beta
\]

\[
\text{Predict} (X \rightarrow \beta) = \text{First} (\beta) \\
= \beta
\]

Similarly argument for existence of an intersection if \( \beta \) is nullable.
Factoring Out Left Recursion

Transform:

\[ X \rightarrow X\alpha \]
\[ \rightarrow \beta \]

into:

\[ X \rightarrow \beta X_{tail} \]
\[ X_{tail} \rightarrow \alpha X_{tail} \]
\[ \rightarrow \lambda \]
Next Week

- More on practical parser construction
- Bottom up parsing
- LR(1) grammars