Lexical Analysis

Syntax Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Third homework has been posted. Due date: Tuesday, February 26

• First project has been posted. Due dates: CODE Monday March 4; REPORT Wednesday March 6

• Midterm: Wednesday, March 13, in class; closed book, closed notes, 80 minutes
Start with a regular expression

\[ r_0 \mid r_1 \mid r_2 \mid r_3 \mid r_4 \mid r_5 \mid r_6 \mid r_7 \mid r_8 \mid r_9 \]

*The Cycle of Constructions*
Thompson’s construction produces

The Cycle of Constructions
Abbreviated Register Specification

The subset construction builds

This is a DFA, but it has a lot of states ...

The Cycle of Constructions
The DFA minimization algorithm builds

![DFA diagram]

This looks like what a skilled compiler writer would do!

*The Cycle of Constructions*
Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

\[
\begin{align*}
\text{Term} & \rightarrow [a-zA-Z] ([a-zA-z] | [0-9])^* \\
\text{Op} & \rightarrow \pm | \cdot | * | / \\
\text{Expr} & \rightarrow (\text{Term Op})^* \text{ Term}
\end{align*}
\]

Of course, this would generate a DFA ...

If REs are so useful ...

Why not use them for everything?
Limits of Regular Languages

Not all languages are regular:
\( \text{RL's} \subset \text{CFL's} \subset \text{CSL's} \)

You cannot construct DFA’s to recognize these languages:

- \( L = \{ p^k q^k \} \) (parenthesis languages)
- \( L = \{ wcw^r | w \in \Sigma^* \} \)

Neither of these is a regular language.

But, this is a little subtle. You can construct DFA’s for:

- Strings with alternating 0’s and 1’s
  \( (\varepsilon | 1)(01)^*(\varepsilon | 0) \)
- Strings with and even number of 0’s and 1’s
- Strings of bit patterns that represent binary numbers which are divisible by 5 (homework)
Poor language design can complicate scanning

- **Reserved words are important**
  
  \[\text{if then then then = else; else else = then}\]  
  \(\text{(PL/I)}\)

- **Insignificant blanks**
  
  \[\text{do 10 i = 1,25}\]  
  \[\text{do 10 i = 1.25}\]  
  \(\text{(Fortran & Algol68)}\)

- **String constants with special characters**
  
  newline, tab, quote, comment delimiters, …  
  \(\text{(C, C++, Java, …)}\)

- **Limited identifier “length”**
  
  \(\text{(Fortran 66 & PL/I)}\)
Parsing
(Syntax Analysis)

EAC Chapters 3.1 - 3.2
Parser

- Checks the stream of *words* and their *parts of speech* (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code
The process of discovering a derivation for some sentence

- Need a mathematical model of syntax — a grammar $G$
- Need an algorithm for testing membership in $L(G)$
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap

1. Context-free grammars and derivations
2. Top-down parsing
   - LL(1) parsers, hand-coded recursive descent parsers
3. Bottom-up parsing
   - Automatically generated LR(1) parsers
Context-free syntax is specified with a context-free grammar

SheepNoise → SheepNoise  baa
|  baa

This CFG defines the set of noises sheep normally make

It is written in a variant of Backus-Naur form

Formally, a grammar is a four tuple, \( G = (S,N,T,P) \)

- \( S \) is the start symbol \((\text{set of strings in } L(G))\)
- \( N \) is a set of non-terminal symbols \( (\text{syntactic variables}) \)
- \( T \) is a set of terminal symbols \( (\text{words}) \)
- \( P \) is a set of productions or rewrite rules \( (P: N \rightarrow (N \cup T)^*) \)

\[ L(G) = \{ w \in T^* \mid S \Rightarrow^* w \} \]
We can use the *SheepNoise* grammar to create sentences

→ use the productions as *rewriting rules*

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td><em>SheepNoise</em></td>
</tr>
<tr>
<td>2</td>
<td><em>baa</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td><em>SheepNoise</em></td>
</tr>
<tr>
<td>1</td>
<td><em>SheepNoise</em> <em>baa</em></td>
</tr>
<tr>
<td>1</td>
<td><em>SheepNoise</em> <em>baa</em> <em>baa</em></td>
</tr>
<tr>
<td>2</td>
<td><em>baa</em> <em>baa</em> <em>baa</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td><em>SheepNoise</em></td>
</tr>
<tr>
<td>1</td>
<td><em>SheepNoise</em> <em>baa</em></td>
</tr>
<tr>
<td>2</td>
<td><em>baa</em> <em>baa</em></td>
</tr>
</tbody>
</table>

*And so on …*
To explore the uses of CFGs, we need a more complex grammar $G$:

- Such a sequence of rewrites is called a *derivation*.
- Process of discovering a derivation is called *parsing*.

We denote this derivation: $Expr \Rightarrow^* id - num \ast id$
Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest
- **Leftmost derivation** — replace leftmost NT at each step;
  generates left sentential forms ($\Rightarrow^*_{lm}$)
- **Rightmost derivation** — replace rightmost NT at each step;
  generates right sentential forms ($\Rightarrow^*_{rm}$)

These are the two systematic derivations

*(We don’t care about randomly-ordered derivations!)*

The example on the preceding slide was a *leftmost* derivation
- Of course, there is also a *rightmost* derivation
- Interestingly, the resulting parse trees may be different
Rule in our grammar:

\[ E \rightarrow E \text{ Op } E \]

A single derivation step

... \( E \) ...

\[ \Rightarrow \]

... \( E \text{ Op } E \)... 

can be represented as a tree structure with the left-hand side non-terminal as the root, and all right-hand side symbols as the children (ordered left to right).

The entire derivation of a sentence in the language can be represented as a parse tree with the start symbol as its root, and leave nodes that are all terminal symbols.

**NOTE:** The structure of the parse tree has semantic significance!
<sentence> ::= <subject> <verb> <rest>

Example:

time flies like an arrow
fruit flies like a banana.
The Two Derivations for $x - 2 * y$

In both cases, $Expr \Rightarrow^* id - num * id$

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Expr Op Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id,x&gt; Op Expr$</td>
</tr>
<tr>
<td>5</td>
<td>$&lt;id,x&gt; - Expr$</td>
</tr>
<tr>
<td>1</td>
<td>$&lt;id,x&gt; - Expr Op Expr$</td>
</tr>
<tr>
<td>2</td>
<td>$&lt;id,x&gt; - &lt;num2&gt; Op Expr$</td>
</tr>
<tr>
<td>6</td>
<td>$&lt;id,x&gt; - &lt;num2&gt; * Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id,x&gt; - &lt;num2&gt; * &lt;id,y&gt;$</td>
</tr>
</tbody>
</table>

**Leftmost derivation**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Expr Op Expr$</td>
</tr>
<tr>
<td>3</td>
<td>$Expr Op &lt;id,y&gt;$</td>
</tr>
<tr>
<td>6</td>
<td>$Expr * &lt;id,y&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr Op Expr * &lt;id,y&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>$Expr Op &lt;num2&gt; * &lt;id,y&gt;$</td>
</tr>
<tr>
<td>5</td>
<td>$Expr - &lt;num2&gt; * &lt;id,y&gt;$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id,x&gt; - &lt;num2&gt; * &lt;id,y&gt;$</td>
</tr>
</tbody>
</table>

**Rightmost derivation**
Leftmost derivation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>Expr</td>
</tr>
<tr>
<td>1</td>
<td>Expr Op Expr</td>
</tr>
<tr>
<td>3</td>
<td>(&lt;\text{id},x&gt;) Op Expr</td>
</tr>
<tr>
<td>5</td>
<td>(&lt;\text{id},x&gt;) - Expr</td>
</tr>
<tr>
<td>1</td>
<td>(&lt;\text{id},x&gt;) - Expr Op Expr</td>
</tr>
<tr>
<td>2</td>
<td>(&lt;\text{id},x&gt;) - (&lt;\text{num},2&gt;) Op Expr</td>
</tr>
<tr>
<td>6</td>
<td>(&lt;\text{id},x&gt;) - (&lt;\text{num},2&gt;) * Expr</td>
</tr>
<tr>
<td>3</td>
<td>(&lt;\text{id},x&gt;) - (&lt;\text{num},2&gt;) * (&lt;\text{id},y&gt;)</td>
</tr>
</tbody>
</table>

This evaluates as \(x - (2 \ast y)\)
This corresponds to our rightmost derivation.
Can we get this with another leftmost derivation as well?

This evaluates as \((x - 2) * y\)
Two Leftmost Derivations for $x - 2 * y$

The Difference:

- Different productions chosen on the second step

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Original choice</th>
<th>Rule</th>
<th>Sentential Form</th>
<th>New choice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Expr$</td>
<td></td>
<td></td>
<td>$Expr$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$Expr \ Op \ Expr$</td>
<td></td>
<td>1</td>
<td>$Expr \ Op \ Expr$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id, x&gt; \ Op \ Expr$</td>
<td></td>
<td>1</td>
<td>$Expr \ Op \ Expr \ Op \ Expr$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$&lt;id, x&gt; - \ Expr$</td>
<td></td>
<td>3</td>
<td>$&lt;id, x&gt; \ Op \ Expr \ Op \ Expr$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$&lt;id, x&gt; - \ Expr \ Op \ Expr$</td>
<td></td>
<td>5</td>
<td>$&lt;id, x&gt; - \ Expr \ Op \ Expr$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; \ Op \ Expr$</td>
<td></td>
<td>2</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; \ Op \ Expr$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; * \ Expr$</td>
<td></td>
<td>6</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; * \ Expr$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; * &lt;id, y&gt;$</td>
<td></td>
<td>3</td>
<td>$&lt;id, x&gt; - &lt;num, 2&gt; * &lt;id, y&gt;$</td>
<td></td>
</tr>
</tbody>
</table>

- Both derivations succeed in producing $x - 2 * y$
These two derivations point out a problem with the grammar. How to resolve ambiguity?
Answer: Change grammar to enforce operator precedence and associativity

To add precedence
• Create a non-terminal for each *level of precedence*
• Isolate the corresponding part of the grammar
• Force the parser to recognize high precedence subexpressions first

For algebraic expressions
• *Multiplication and division, first* (level one)
• *Subtraction and addition, next* (level two)

Note: we are ignoring the issue of associativity for now
Adding the standard algebraic precedence produces:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Expr</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr → Expr + Term</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Expr - Term</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Term</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Term → Term * Factor</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This grammar is slightly larger
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses \( x - 2 \times y \)
Derivations and Precedence

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
</tr>
<tr>
<td>3</td>
<td>Expr - Term</td>
</tr>
<tr>
<td>5</td>
<td>Expr - Term * Factor</td>
</tr>
<tr>
<td>9</td>
<td>Expr - Term * &lt;id,y&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Expr - Factor * &lt;id,y&gt;</td>
</tr>
<tr>
<td>8</td>
<td>Expr - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>4</td>
<td>Term - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>7</td>
<td>Factor - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

The rightmost derivation

This produces \( x - (2 * y) \), along with an appropriate parse tree.
Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.
Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is **ambiguous**
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is **ambiguous**
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the *if-then-else* problem

\[
Stmt \rightarrow \text{if } Expr \text{ then } Stmt \\
| \text{if } Expr \text{ then } Stmt \text{ else } Stmt \\
| \text{... other stmts ...}
\]

*This ambiguity is entirely grammatical in nature*
Ambiguity

This sentential form has two derivations

\[
\text{if } \text{Expr}_1 \text{ then if } \text{Expr}_2 \text{ then } \text{Stmt}_1 \text{ else } \text{Stmt}_2
\]
Ambiguity

Removing the ambiguity

- **Must rewrite the grammar to avoid generating the problem**
- **Match each `else` to innermost unmatched `if`** *(common sense rule)*

1. $Stmt \rightarrow \text{WithElse}$
2. \hspace{1em} $|$ $\text{NoElse}$
3. $\text{WithElse} \rightarrow \text{if Expr then WithElse else WithElse}$
4. \hspace{1em} $|$ $\text{OtherStmt}$
5. $\text{NoElse} \rightarrow \text{if Expr then Stmt}$
6. \hspace{1em} $|$ $\text{if Expr then WithElse else NoElse}$

With this grammar, the example has only one derivation
### Ambiguity

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Stmt</td>
</tr>
<tr>
<td>2</td>
<td>NoElse</td>
</tr>
<tr>
<td>5</td>
<td>if Expr then Stmt</td>
</tr>
<tr>
<td>?</td>
<td>if E₁ then Stmt</td>
</tr>
<tr>
<td>1</td>
<td>if E₁ then WithElse</td>
</tr>
<tr>
<td>3</td>
<td>if E₁ then if Expr then WithElse else WithElse</td>
</tr>
<tr>
<td>?</td>
<td>if E₁ then if E₂ then WithElse else WithElse</td>
</tr>
<tr>
<td>4</td>
<td>if E₁ then if E₂ then S₁ else WithElse</td>
</tr>
<tr>
<td>4</td>
<td>if E₁ then if E₂ then S₁ else S₂</td>
</tr>
</tbody>
</table>

This binds the `else` controlling `S₂` to the inner `if`
Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

\[ a = f(17) \]

In many Algol-like languages, \( f \) could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of type, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - Step outside grammar rather than use a more complex grammar
Ambiguity arises from two distinct sources
• Confusion in the context-free syntax
  
  (if-then-else)
• Confusion that requires context to resolve
  
  (overloading)

Resolving ambiguity
• To remove context-free ambiguity, rewrite the grammar
• Change language (e.g.: if ... endif)
• To handle context-sensitive ambiguity takes cooperation
  
  → Knowledge of declarations, types, ...
  → Accept a superset of $L(G)$ & check it by other means†
  → This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar
• Parsing techniques that “do the right thing”
• i.e., always select the same derivation

†See Chapter 4
Parsing
(Syntax Analysis)

Top-Down Parsing
EAC Chapters 3.3
LL(1), recursive descent

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

\[ S \Rightarrow_{lm}^* xA \beta \Rightarrow_{lm} x\delta \beta \Rightarrow_{lm}^* xy \]
**LL(1), recursive descent**

1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

```
S ⇒^*lm x A β ⇒_lm x δ β ⇒^*lm x y
```

rule $A \rightarrow \delta$
Remember the expression grammar?

Version with precedence

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal $\rightarrow$ Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr $\rightarrow$ Expr + Term</td>
</tr>
<tr>
<td>3</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>Term $\rightarrow$ Term * Factor</td>
</tr>
<tr>
<td>6</td>
<td>$</td>
</tr>
<tr>
<td>7</td>
<td>$</td>
</tr>
<tr>
<td>8</td>
<td>Factor $\rightarrow$ number</td>
</tr>
<tr>
<td>9</td>
<td>$</td>
</tr>
</tbody>
</table>

And the input $x - 2 * y$
Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left recursive if \( \exists A \in NT \text{ such that } \exists \text{ a derivation } A \Rightarrow^+ A\alpha, \text{ for some string } \alpha \in (NT \cup T)^+ \)

Our expression grammar is left recursive
- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler
To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[ Fee \rightarrow Fee \alpha \]
\[ \quad \mid \beta \]

where neither \( \alpha \) nor \( \beta \) start with \( Fee \)

We can rewrite this as

\[ Fee \rightarrow \beta Fie \]
\[ Fie \rightarrow \alpha Fie \]
\[ \quad \mid \varepsilon \]

where \( Fie \) is a new non-terminal

This accepts the same language, but uses only right recursion
The expression grammar contains two cases of left recursion

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Expr} \ + \ \text{Term} & \text{Term} & \rightarrow \text{Term} \ * \ \text{Factor} \\
& | \ \text{Expr} \ - \ \text{Term} & | \ \text{Term} \ / \ \text{Factor} \\
& | \ \text{Term} & | \ \text{Factor}
\end{align*}
\]

Applying the transformation yields

\[
\begin{align*}
\text{Expr} & \rightarrow \ \text{Term} \ \text{Expr}' \\
\text{Expr}' & | \ + \ \text{Term} \ \text{Expr}' \\
& | \ - \ \text{Term} \ \text{Expr}' \\
& | \ \varepsilon
\end{align*}
\]

\[
\begin{align*}
\text{Term} & \rightarrow \ \text{Factor} \ \text{Term}' \\
\text{Term}' & | \ * \ \text{Factor} \ \text{Term}' \\
& | \ / \ \text{Factor} \ \text{Term}' \\
& | \ \varepsilon
\end{align*}
\]

These fragments use only right recursion
Substituting them back into the grammar yields

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal $\rightarrow$ Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr $\rightarrow$ Term Expr'</td>
</tr>
<tr>
<td>3</td>
<td>Expr' $\rightarrow$ + Term Expr'</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Term $\rightarrow$ Factor Term'</td>
</tr>
<tr>
<td>7</td>
<td>Term' $\rightarrow$ * Factor</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Factor $\rightarrow$ number</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- This grammar is correct, if somewhat non-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- **General left recursion removal algorithm p.103 EAC**
We set out to study parsing

- Specifying syntax
  - Context-free grammars
  - Ambiguity

- Top-down parsers
  - Algorithm & its problem with left recursion
  - Left-recursion removal
  - Left factoring (will discuss later)

- Predictive top-down parsing
  - The LL(1) condition
  - Table-driven LL(1) parsers
  - Recursive descent parsers
    - Syntax directed translation (example)
If it picks the wrong production, a top-down parser may backtrack. An alternative is to look ahead in input and use context to pick correctly.

How much lookahead is needed?
- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley’s algorithm

Fortunately,
- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are $LL(1)$ and $LR(1)$ grammars.
Basic idea

*Given* $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between $\alpha$ & $\beta$.

**FIRST sets**

For some *rhs* $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first (terminal) symbol in some string that derives from $\alpha$.

That is, $a \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$. 
The FIRST Set - 1 symbol lookahead

\[ a \in \text{FIRST}_{1}(\alpha) \iff \alpha \Rightarrow^{*} a \gamma, \text{ for some } \gamma \]

To build FIRST(X) for all grammar symbols X:

1. if X is a terminal (token), FIRST(X) := \{ X \}
2. if X → ε, then ε ∈ FIRST(X)
3. iterate until no more terminals or ε can be added to any FIRST(X):
   
   if \ X \to Y_1 Y_2 ... Y_k \ then
   
   a ∈ FIRST(X) if a ∈ FIRST(Y_i) and
   
   ε ∈ FIRST(Y_j) for all 1 ≤ j < i
   
   ε ∈ FIRST(X) if ε ∈ FIRST(Y_i) for all 1 ≤ i ≤ k

end iterate

Note: if ε \not\in FIRST(Y_1), then FIRST(Y_i) is irrelevant, for 1 < i
The FIRST Set

\[ a \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build \( \text{FIRST}(\alpha) \) for \( \alpha = X_1 X_2 \ldots X_n \):

1. \( a \in \text{FIRST}(\alpha) \) if \( a \in \text{FIRST}(X_i) \) and 
   \[ \epsilon \in \text{FIRST}(X_j) \text{ for all } 1 \leq j < i \]

2. \( \epsilon \in \text{FIRST}(\alpha) \) if \( \epsilon \in \text{FIRST}(X_i) \) for all \( 1 \leq i \leq n \)
Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between $\alpha$ & $\beta$

**FIRST** sets

For some rhs $\alpha \in G$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string that derives from $\alpha$

That is, $\alpha \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

This is almost correct, but not quite
More Syntax Analysis (top-down parsing)

Read EaC: Chapter 3.3