Finite-State Automata

- Alphabet $\Sigma$
- Set of states with initial and accept states
- Transitions between states labeled with letters

$$(0|1)^*.(0|1)^*$$

- Green state
- Blue state
Automaton

• String recognition
  – Start with start state and first letter of string
  – At each step, match current letter against a transition whose label is same as letter
  – Continue until reach end of string or match fails
  – If end in accept state, automaton accepts string

• The language of automaton is the set of strings it accepts
Example

Current state

1 1 0

Start state

Accept state

1 1 . 0
Example

Current state

1
0

11.0

1
0

Start state

Accept state
Example

Current state

```
1
1
0
0
```

1 1 . 0

Start state

Accept state
Example

Current state

1
0

1 1 . 0

Start state

Accept state

1 1 . 0
Example

Current state

Start state

Accept state

String is accepted!

1 1 . 0

1 1 0

Current letter

Current state

String is accepted!
Generation Versus Recognition

• Syntactically a language is a set of strings
• Regular expressions
  – Defining a language by composition
• Automata
  – Defining a language by implementation
    • Determining whether a string is in the language or not
  – Theoretically equivalent (for regular expressions and automata)

Can we compile the Regular Expression Language into an Automata Implementation?

Yes we can!
NFA vs. DFA

• DFA
  – No $\varepsilon$ transitions
  – At most one transition from each state for each letter

• NFA – can do both
  – It can be in many states at the same time
  – accept if ANY of its current states is an accepting state
RE to Automata

- Construction by structural induction

- Convert regular expression into an automaton with
  - One start state
  - One accept state

- Assume any sub-expressions have been converted
  - this works as long as you do it bottom up
RE to Automata

\[ \epsilon \in \Sigma \]

- Start state
- Accept state
Sequence
Sequence
Sequence
Sequence
Sequence

$\epsilon$ $r_1$ $r_2$

Old start state Start state
Old accept state Accept state
Choice

\[ r_1 | r_2 \]

- Start state
- Accept state
Choice

- Old start state
- Old accept state
- Start state
- Accept state

\[ r_1 | r_2 \]

\[ r_1 \]

\[ r_2 \]
Choice

Old start state  Start state
Old accept state  Accept state

\( r_1 \) \( r_2 \)

\( \epsilon \) \( \epsilon \)
Choice

- Old start state
- Start state
- Old accept state
- Accept state

\[ r_1 | r_2 \]

\[ \varepsilon \]

\[ r_1 \]

\[ \varepsilon \]

\[ r_2 \]

\[ \varepsilon \]
Kleene Star

Old start state
Old accept state
Start state
Accept state
Kleene Star

- Old start state
- Old accept state
- Start state
- Accept state
Kleene Star

\[ r^* \]

- Old start state
- Old accept state
- Start state
- Accept state
Kleene Star
Kleene Star

- Old start state
- Start state
- Old accept state
- Accept state
Conversions

- Our regular expression to automata conversion produces an NFA
- Would like to have a DFA to make recognition algorithm simpler
- Can convert from NFA to DFA (but DFA may be exponentially larger than NFA)
  - Simple algorithm available (check a text book for details)
NFA to DFA Construction

- DFA has a state for each subset of states in NFA
  - DFA start state corresponds to set of states reachable by following ε transitions from NFA start state
  - DFA state is an accept state if an NFA accept state is in its set of NFA states
- To compute the transition for a given DFA state D and letter a
  - Set S to empty set
  - Find the set N of D’s NFA states
    - For all NFA states n in N
      - Compute set of states N’ that the NFA may be in after matching a
      - Set S to S union N’
    - If S is nonempty, there is a transition for a from D to the DFA state that has the set S of NFA states
    - Otherwise, there is no transition for a from D
NFA to DFA Example for \((a|b)^*.(a|b)^*\)
Lexical Structure in Languages

• Typical classes of tokens:
  – Keywords (if, while)
  – Arithmetic Operations (+, -, *, /)
  – Integer numbers (1, 2, 45, 67)
  – Floating point numbers (1.0, .2, 3.337)
  – Identifiers (abc, i, j, ab345)

• Typically have a lexical category for each
  – Define each class of tokens by a regular expression

• Also need lexical category for
  – comments
  – whitespace
Lexical Categories Example

- IfKeyword = if
- WhileKeyword = while
- Operator = +|\-|\*|/
- Integer = [0-9] [0-9]*
- Float = [0-9]*. [0-9]*
- Identifier = [a-z](\[a-z\]|\[0-9\])*  
- Note that  
  \[0-9\] = (0|1|2|3|4|5|6|7|8|9)  
  \[a-z\] = (a|b|c|...|y|z)
- Will use lexical categories in next level
Write a Regular Expression

• All strings of the wedge alphabet \{ <, > \}
  – ( <|> )* 

• Strings with open wedges followed by close wedges
  – <*>*

• Strings with matching wedges
  – Not with a regular expression!
Nested Expressions

• Are regular languages sufficient for specifying syntax?

• Try the following
  – (a+(b-c))*(d-(x-(y-z)))
  – if (x < y) if (y < z) a = 5 else a = 6 else a = 7
Context-Free Grammar

- Set of terminals
  \{ Op, Int, Open, Close \}
  Each terminal defined
  by regular expression

- Set of nonterminals
  \{ Start, Expr \}

- Set of productions
  - Single nonterminal on LHS
  - Sequence of terminals and nonterminals on RHS

\begin{align*}
Op &= +|-|\ast|/ \\
Int &= \{0-9\} \{0-9\}^* \\
Open &= < \\
Close &= > \\
Start &\rightarrow Expr \\
Expr &\rightarrow Expr \text{ Op } Expr \\
Expr &\rightarrow Int \\
Expr &\rightarrow Open \text{ Expr } Close
\end{align*}
Generation

start with *Start* nonterminal
repeat
  choose a nonterminal
  choose a production with that nonterminal in LHS
  replace nonterminal with RHS of production
until no more nonterminals in the string
Sample Derivation

Op = +|-|*|/
Int = [0-9] [0-9]*
Open = <
Close = >

1) Start → Expr
2) Expr → Expr Op Expr
3) Expr → Int
4) Expr → Open Expr Close

Open Expr Close Op Expr
Open Expr Op Expr Close Op Expr
Open Int Op Expr Close Op Expr
Open Int Op Expr Close Op Int
Open Int Op Int Close Op Int
Sample Derivation

Op = +|-|*|/
Int = [0-9] [0-9]*
Open = <
Close = >

1) \textit{Start} \rightarrow \textit{Expr}
2) \textit{Expr} \rightarrow \textit{Expr} \textit{Op} \textit{Expr}
3) \textit{Expr} \rightarrow \textit{Int}
4) \textit{Expr} \rightarrow \textit{Open} \textit{Expr} \textit{Close}
Parse Tree

- Tracks the history of a derivation
- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges: individual derivations
Parse Tree for \(<2-1>+1\)
**Parsing**

Op = +|-|*|/
Int = [0-9] [0-9]*
Open = <
Close = >

1) **Start** → **Expr**
2) **Expr** → **Expr** Op **Expr**
3) **Expr** → Int
4) **Expr** → Open **Expr** Close

< 2 - 1 > + 1
Open 2 - 1 > + 1
Open Int - 1 > + 1
Open Int Op 1 > + 1
Open Int Op Int > + 1
Open Int Op Int Close + 1
Open Int Op Int Close Op 1
Open Int Op Int Close Op Int
Parsing

Op = +|-|*|/
Int = [0-9] [0-9]*
Open = <
Close = >

1) Start → Expr
2) Expr → Expr Op Expr
3) Expr → Int
4) Expr → Open Expr Close
What is this Grammar?

What is the language of the following grammar?

Start $\rightarrow$ S
S $\rightarrow$ ( L )
S $\rightarrow$ a
L $\rightarrow$ L , S
L $\rightarrow$ S
Start → S
S → ( L )
S → a
L → L , S
L → S

Write a parse tree for
(a, (a, a))
Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques
Terminology

• Many different parsing techniques
  – Each can handle some set of CFGs
  – Categorization of techniques
• Many different parsing techniques
  – Each can handle some set of CFGs
  – Categorization of techniques

  – \( L \) - parse from left to right
  – \( R \) - parse from right to left
Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

  - \textbf{L} - leftmost derivation
  - \textbf{R} - rightmost derivation
Terminology

• Many different parsing techniques
  – Each can handle some set of CFGs
  – Categorization of techniques
  – Number of lookahead characters
• Many different parsing techniques
  – Each can handle some set of CFGs
  – Categorization of techniques
  – Examples: LL(0), LR(1)

• Today: Building a LL(k) parser
  – Manual construction of a recursive descent parser
    • Code parser as set of mutually recursive procedures
    • Structure of parser matches structure of grammar
      $LR(k)$
Ambiguity
Ambiguity in Grammar

• Grammar is ambiguous if there are multiple derivations (therefore multiple parse trees) for a single string

• Derivation and parse tree usually reflect semantics of the program

• Ambiguity in grammar often reflects ambiguity in semantics of language (which is considered undesirable)
Two parse trees for 2-1+1

Tree corresponding to <2-1>+1

Tree corresponding to 2-<1+1>

Start

Expr

Expr

Op

Expr

Int

2

1

2

1
Eliminating Ambiguity

Solution: hack the grammar

Original Grammar

\[
\text{Start} \rightarrow \text{Expr} \\
\text{Expr} \rightarrow \text{Expr} \text{ Op} \text{ Expr} \\
\text{Expr} \rightarrow \text{Int} \\
\text{Expr} \rightarrow \text{Open} \text{ Expr} \text{ Close}
\]

Hacked Grammar

\[
\text{Start} \rightarrow \text{Expr} \\
\text{Expr} \rightarrow \text{Expr} \text{ Op} \text{ Int} \\
\text{Expr} \rightarrow \text{Int} \\
\text{Expr} \rightarrow \text{Open} \text{ Expr} \text{ Close}
\]

Conceptually, makes all operators associate to left
Parse Trees for Hacked Grammar

Only one parse tree for 2-1+1!

Valid parse tree

No longer valid parse tree
Exercise

• Construct a grammar for this language:

Expressions involving Ints, +, -, *, /
Precedence Violations

- All operators associate to left
- Violates precedence of * over +
  - $2-3*4$ associates like $<2-3>*4$

```
Start
    ↓
Expr
    ↓
Expr  Op  Int
    ↓  *  4
Int  3
    ↓
Int  2
```
## Hacking Around Precedence

<table>
<thead>
<tr>
<th>Original Grammar</th>
<th>Hacked Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op</strong> = +</td>
<td>-</td>
</tr>
<tr>
<td><strong>Int</strong> = [0-9] [0-9]*</td>
<td><strong>MulOp</strong> = *</td>
</tr>
<tr>
<td><strong>Open</strong> = &lt;</td>
<td><strong>Int</strong> = [0-9] [0-9]*</td>
</tr>
<tr>
<td><strong>Close</strong> = &gt;</td>
<td><strong>Open</strong> = &lt;</td>
</tr>
<tr>
<td><strong>Close</strong> = &gt;</td>
<td></td>
</tr>
</tbody>
</table>

**Start** → **Expr**

**Expr** → **Expr** **Op** **Int**

**Expr** → **Int**

**Expr** → **Open** **Expr** **Close**

**Start** → **Expr**

**Expr** → **Expr** **AddOp** **Term**

**Expr** → **Term**

**Term** → **Term** **MulOp** **Num**

**Term** → **Num**

**Num** → **Int**

**Num** → **Open** **Expr** **Close**
Parse Tree Changes

Old parse tree for 2-3*4

Start
  └── Expr
    └── Expr
        └── Expr
            └── Int 2
        └── Op
            └── Int 3
    └── Op
        └── *
            └── Int 4

New parse tree for 2-3*4

Start
  └── Expr
    └── AddOp
        └── Term
            └── MulOp
                └── Num
                    └── Int 4
            └── Num
                └── Int 3
        └── Term
            └── Num
                └── Int 2
General Idea

• Group Operators into Precedence Levels
  – * and / are at top level, bind strongest
  – + and - are at next level, bind next strongest

• Nonterminal for each Precedence Level
  – Term is nonterminal for * and /
  – Expr is nonterminal for + and -

• Can make operators left or right associative within each level

• Generalizes for arbitrary levels of precedence
## What is different?

<table>
<thead>
<tr>
<th>Hacked Grammar I</th>
<th>Hacked Grammar II</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddOp = +</td>
<td>-</td>
</tr>
<tr>
<td>MulOp = *</td>
<td>/</td>
</tr>
<tr>
<td>Int = [0-9] [0-9]*</td>
<td>Int = [0-9] [0-9]*</td>
</tr>
<tr>
<td>Open = &lt;</td>
<td>Open = &lt;</td>
</tr>
<tr>
<td>Close = &gt;</td>
<td>Close = &gt;</td>
</tr>
<tr>
<td><strong>Start</strong> → <strong>Expr</strong></td>
<td><strong>Start</strong> → <strong>Expr</strong></td>
</tr>
<tr>
<td><strong>Expr</strong> → <strong>Expr</strong> <strong>AddOp</strong> <strong>Term</strong></td>
<td><strong>Expr</strong> → <strong>Expr</strong> <strong>AddOp</strong> <strong>Term</strong></td>
</tr>
<tr>
<td><strong>Expr</strong> → <strong>Term</strong></td>
<td><strong>Expr</strong> → <strong>Term</strong></td>
</tr>
<tr>
<td><strong>Term</strong> → <strong>Term</strong> <strong>MulOp</strong> <strong>Num</strong></td>
<td><strong>Term</strong> → <strong>Term</strong> <strong>MulOp</strong> <strong>Num</strong></td>
</tr>
<tr>
<td><strong>Term</strong> → <strong>Num</strong></td>
<td><strong>Term</strong> → <strong>Num</strong></td>
</tr>
<tr>
<td><strong>Num</strong> → <strong>Int</strong></td>
<td><strong>Num</strong> → <strong>Int</strong></td>
</tr>
<tr>
<td><strong>Num</strong> → <strong>Open</strong> <strong>Expr</strong> <strong>Close</strong></td>
<td><strong>Num</strong> → <strong>Int</strong></td>
</tr>
</tbody>
</table>
Handling If Then Else

\[
\begin{align*}
Start & \rightarrow Stat \\
Stat & \rightarrow \text{if} \; Expr \; \text{then} \; Stat \; \text{else} \; Stat \\
Stat & \rightarrow \text{if} \; Expr \; \text{then} \; Stat \\
Stat & \rightarrow \ldots
\end{align*}
\]
Parse Trees

• Consider Statement
  if $e_1$ then if $e_2$ then $s_1$ else $s_2$
Two Parse Trees

Which is correct?
Alternative Readings

- **Parse Tree Number 1**
  
  ```
  if \( e_1 \)
  
  if \( e_2 \)
    \( s_1 \)
  
  else
    \( s_2 \)
  
  \( \leftarrow \) Programming languages prefer this
  ```

- **Parse Tree Number 2**
  
  ```
  if \( e_1 \)
  
  if \( e_2 \)
    \( s_1 \)
  
  else
    \( s_2 \)
  ```

Grammar is ambiguous

This is known as the dangling-else problem
Resolving the dangling-else

- Rule: Associate else with the closest if
  - Suppose you have `If e then S1 else S2`
    - Can `S1 = If e then S3`?
    - NO! That would violate our rule.
    - An `ifStmt` inside `S1` must have an else part
      - (Or be inside a block)
  - This can be encoded in the grammar.
Hacked Grammar

\[
\begin{align*}
    \text{Start} & \rightarrow \text{Stmt} \\
    \text{Stmt} & \rightarrow \text{if } \text{Expr} \text{ then } \text{Stmt} \\
    \text{Stat} & \rightarrow \text{if } \text{Expr} \text{ then } \text{ThenStmt} \text{ else } \text{Stmt} \\
    \text{Stmt} & \rightarrow \text{OtherStmt} \\
    \text{ThenStmt} & \rightarrow \text{if } \text{Expr} \text{ then } \text{ThenStmt} \text{ else } \text{ThenStmt} \\
    \text{ThenStmt} & \rightarrow \text{OtherStmt}
\end{align*}
\]
Hacked Grammar

- Basic Idea: control carefully where an if without an else can occur
  - Either at top level of statement
  - Or as very last in a sequence of if then else if then ... statements
Abstract Syntax
Trees
Abstract Versus Concrete Trees

• Remember grammar hacks
  – left factoring, ambiguity elimination, precedence of binary operators

• Hacks lead to a tree that may not reflect cleanest interpretation of program

• May be more convenient to work with abstract syntax tree (roughly, parse tree from grammar before hacks)
Example

\[ x^2 + y^2 \]

Each node in the tree is a data structure. Immutability can allow sharing.
Example

class eBinary : public Expression {
protected:
    const Expression* left_;  
    const Expression* right_; 
    const BinaryOperator op_; 

public:
    eBinary(Expression* left, Expression* right, BinaryOperator op):
    Expression(Expression::BINOP),   
               left_(left), right_(right), op_(op) {} 
    const Expression* left() {   return left_; } 
    const Expression* right() {   return right_; } 
    const BinaryOperator op() {   return op_; } 
    void accept(Visitor& v) {
        v.visit(*this); 
    } 
};
Top Down Parsing
Basic Approach

• Start with Start symbol
• Build a leftmost derivation
  – If leftmost symbol is nonterminal, choose a production and apply it
  – If leftmost symbol is terminal, match against input
  – If all terminals match, have found a parse!
  – Key: find correct productions for nonterminals
Graphical Illustration of Leftmost Derivation

Sentential Form

\[ NT_1 \ T_1 \ T_2 \ T_3 \ NT_2 \ NT_3 \]

Apply Production Here

Not Here
Grammar for Parsing Example

\[
\begin{align*}
\text{Start} & \rightarrow \text{Expr} \\
\text{Expr} & \rightarrow \text{Expr} + \text{Term} \\
\text{Expr} & \rightarrow \text{Expr} - \text{Term} \\
\text{Expr} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{Term} \times \text{Int} \\
\text{Term} & \rightarrow \text{Term} / \text{Int} \\
\text{Term} & \rightarrow \text{Int}
\end{align*}
\]

- Set of tokens is \{ +, -, *, /, Int \}, where Int = [0-9][0-9]*
  - For convenience, may represent each Int n token by n
Parsing Example

Parse Tree

Start

Remaining Input

2-2*2

Sentential Form

Start

Current Position in Parse Tree
 Parsing Example

Start → Expr

Remaining Input
2-2*2

Sentential Form
Expr

Applied Production
Start → Expr

Current Position in Parse Tree

Parse Tree

Start

Expr
Parsing Example

Parse Tree

Remaining Input
2 - 2 * 2

Sentential Form
Expr - Term

Applied Production
Expr → Expr - Term

Expr → Expr + Term

Expr → Expr - Term

Expr → Term

Saman Amarasinghe

6.035  ©MIT Fall 2006
Parsing Example

Parse Tree

remaining input

2-2*2

sentential form

Term - Term

applied production

Expr → Term

Expr → Expr + Term

Expr → Expr - Term

expr

term

term

start

expr

expr - term

expr

expr → term

expr → term + term

expr → term - term
**Parsing Example**

Remaining Input:

2-2*2

Sentential Form:

Int - Term

Applied Production:

Term → Int

Parse Tree:

```
Start
  /\   \\
Expr -   Term
  | |  |
Expr - Int
```

Parse Tree:

```
Start
  /\   \\
Expr -   Term
  | |  |
Expr - Int
```

Start

Expr

Term

Int
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Term

Int 2

Remaining Input

2-2*2

Sentential Form

2 - Term

Match Input Token!
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Term

Int 2

Remaining Input

-2*2

Sentential Form

2 - Term

Match Input Token!
Parsing Example

Parse Tree

Remaining Input

Sentential Form

-2*2

2 - Term
Parsing Example

Parse Tree

```
Start
  \|-- Expr
    \|-- Term
      \|-- Int 2
```

Remaining Input

```
2*2
```

Sentential Form

```
2 - Term
```
Parsing Example

Parse Tree

Remaining Input

Sentential Form

Expr

Start

2*2

2 - Term

Term

Expr

Term

Int 2
Parsing Example

**Parse Tree**

- **Start**
  - **Expr**
    - **Expr** - **Term**
      - **Term**
        - **Term**
          - **Int 2**
  - **Term**
    - **Term**
      - *** Int**

**Remaining Input**

- 2*2

**Sentential Form**

- 2 - **Term** *Int

**Applied Production**

- **Term** → **Term** *Int
Parsing Example

Parse Tree

Remaining Input

2*2

Sentential Form

2 - Int * Int

Applied Production

Term → Int
Parsing Example

Remaining Input
2*2

Sentential Form
2 - 2*Int

Parse Tree

Start

Expr

Expr

Term

Term

Term

Int 2

Int 2

Match Input Token!
Parsing Example

Parse Tree

Remaining Input

Sentential Form

2 - 2 * Int
Parsing Example

Parse Tree

Start -> Expr
Expr -> Term
Term -> * Int
Int 2

Remaining Input
*2
Sentential Form
2 - 2 * Int
Parsing Example

Parse Tree

Start

Expr

Expr

- 

Term

Term

* 

Int

Int 2

Int 2

Remaining Input

2

Sentential Form

2 - 2 * Int

Match
Input
Token!
Parsing Example

Parse Tree

Remaining Input

2

Sentential Form

2 - 2 * Int

Match
Input
Token!
Parsing Example

Parse Tree

Start

Expr

Expr

Term

Term

Term

Int 2

Input Token!

Match

Remaining Input

Sentential Form

2 - 2 * Int
Parsing Example

Parse Tree

Start

Expr

- Term

Expr

Term

* Int 2

Term

Int 2

Int 2

Remaining Input

Sentential Form

2 - 2 * 2

Parse Complete!
Summary

- Three Actions (Mechanisms)
  - Apply production to expand current nonterminal in parse tree
  - Match current terminal (consuming input)
  - Accept the parse as correct

- Parser generates preorder traversal of parse tree
  - visit parents before children
  - visit siblings from left to right
Policy Problem

- Which production to use for each nonterminal?
- Classical Separation of Policy and Mechanism
- One Approach: Backtracking
  - Treat it as a search problem
  - At each choice point, try next alternative
  - If it is clear that current try fails, go back to previous choice and try something different

- General technique for searching
- Used a lot in classical AI and natural language processing ( parsing, speech recognition )
Left Recursion + Top-Down Parsing = Infinite Loop

- Example Production: $Term \rightarrow Term^* Num$
- Potential parsing steps:
Eliminating Left Recursion

- Start with productions of form
  - $A \rightarrow A \alpha$
  - $A \rightarrow \beta$
  - $\alpha$, $\beta$ sequences of terminals and nonterminals that do not start with $A$

- Repeated application of $A \rightarrow A \alpha$
  builds parse tree like this:
Eliminating Left Recursion

• Replacement productions
  – A → A α         A → β R       R is a new nonterminal
  – A → β           R → α R
  – R → ε           New Parse Tree

Old Parse Tree

A
  /\       /
 A  α     B
  /\      /\   \
 β  α     α  ε

New Parse Tree

A
 /\        /
β R       α R
 /\      /\   \
 α α     α   ε
Hacked Grammar

Original Grammar

Fragment

Term → Term * Int
Term → Term / Int
Term → Int

New Grammar

Fragment

Term → Int Term’
Term’ → * Int Term’
Term’ → / Int Term’
Term’ → ε
Parse Tree Comparisons

Original Grammar

```
Term
  /  
Term * Int
  /  
Int * Int
```

New Grammar

```
Term
  /  
Int Term'
    /  
  * Int
    /  
  Term'
    /  
  * Int
    /  
  Term'
    /  
  * Int
    /  
  Term'
    /  
    ε
```
Eliminating Left Recursion

• Changes search space exploration algorithm
  – Eliminates direct infinite recursion
  – But grammar less intuitive
• Sets things up for predictive parsing
Predictive Parsing

• Alternative to backtracking
• Useful for programming languages, which can be designed to make parsing easier
• Basic idea
  – Look ahead in input stream
  – Decide which production to apply based on next tokens in input stream
  – We will use one token of lookahead
Predictive Parsing Example

Grammar

\[ \text{Start} \rightarrow \text{Expr} \]
\[ \text{Expr} \rightarrow \text{Term} \text{ Expr}' \]
\[ \text{Expr}' \rightarrow + \text{ Expr}' \]
\[ \text{Expr}' \rightarrow - \text{ Expr}' \]
\[ \text{Expr}' \rightarrow \varepsilon \]

\[ \text{Term} \rightarrow \text{Int} \text{ Term}' \]
\[ \text{Term}' \rightarrow *\text{Int} \text{ Term}' \]
\[ \text{Term}' \rightarrow /\text{Int} \text{ Term}' \]
\[ \text{Term}' \rightarrow \varepsilon \]
Choice Points

• Assume $Term'$ is current position in parse tree
• Have three possible productions to apply
  $Term' \rightarrow * \text{Int } Term'$
  $Term' \rightarrow / \text{Int } Term'$
  $Term' \rightarrow \epsilon$
• Use next token to decide
  – If next token is *, apply $Term' \rightarrow * \text{Int } Term'$
  – If next token is /, apply $Term' \rightarrow / \text{Int } Term'$
  – Otherwise, apply $Term' \rightarrow \epsilon$
Predictive Parsing + Hand Coding = Recursive Descent Parser

- One procedure per nonterminal $NT$
  - Productions $NT \rightarrow \beta_1$, ..., $NT \rightarrow \beta_n$
  - Procedure examines the current input symbol $T$ to determine which production to apply
    - If $T \in \text{First}(\beta_k)$
    - Apply production $k$
    - Consume terminals in $\beta_k$ (check for correct terminal)
    - Recursively call procedures for nonterminals in $\beta_k$
  - Current input symbol stored in global variable token

- Procedures return
  - true if parse succeeds
  - false if parse fails
Bottom Up Parsing