Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 7

• From DSA to Regular Expression
• Top-down parsing
Motivations: exercise 7(b)

• Write a regular expression over the set of symbols \{0,1\} that describes the language of all strings having an even number of 0’s and of 1’s
  – Not easy....
  – A solution: \((00|11)^*\((01|10)(00|11)^*(01|10)(00|11)^*\)^*\)
  – How can we get it?

• Towards the solution: a deterministic automaton accepting the language
• But how do we get the regular expression defining the language accepted by the automaton?
Regular expressions, Automata, and all that...

Regular Expressions

Thompson algorithm

Non-Deterministic Finite Automata

Subset construction

Right-linear (Regular) Grammars

Deterministic Finite Automata

Minimization (Partition/Refinement)
From automata to Regular Expressions

• Three approaches:
  – Dynamic Programming [Scott, Section 2.4 on CD] [Hopcroft, Motwani, Ullman, *Introduction to Automata Theory, Languages and Computation*, Section 3.2.1]
  – Incremental state elimination [HMU, Section 3.2.2]
  – Regular Expression as fixed-point of a continuous function on languages
DFAs and Right-linear Grammars

• In a right-linear (regular) grammar each production is of the form $A \rightarrow wB$ or $A \rightarrow w$ ($w \in T^*$)
• From a DFA to a right-linear grammar

\[
\begin{align*}
A & \rightarrow \varepsilon | 1B | 0D \\
B & \rightarrow 1A | 0C \\
C & \rightarrow 0B | 1D \\
D & \rightarrow 0A | 1C
\end{align*}
\]

• The construction also works for NFA
• A similar construction can transform any right-linear grammar into an NFA (productions might need to be transformed introducing new non-terminals)
Kleene fixed-point theorem

• A complete partial order (CPO) is a partial order with a least element \( \bot \) and such that every increasing chain has a supremum

• Theorem: Every continuous function \( F \) over a complete partial order (CPO) has a least fixed-point, which is the supremum of chain

\[
F(\bot) \leq F(F(\bot)) \leq \ldots \leq F^n(\bot) \leq \ldots
\]
Context Free grammars as functions on the CPO of languages

- Languages over \( \Sigma \) form a complete partial order under set inclusion

- A context free grammar defines a continuous function over (tuples of) languages
  
  \[ \text{A} \rightarrow a \mid bA \quad F(L) = \{a\} \cup \{bw \mid w \in L\} \]

- The language generated by the grammar is the least-fixed point of the associated function
  
  \[ \emptyset \subset \{a\} \subset \{a,ba\} \subset \{a,ba,bba\} \subset \ldots \subset \{b^n a \mid n \geq 0\} \]

- In the case of right-linear grammars we can describe the least fixed-point as a regular expression
  
  \[ \text{Lang(A)} = b^*a \]
Example: from right-linear grammar to regular expression

1) Substitute D in A and C
A → ε | 1B | 0(0A | 1C)
B → 1A | 0C
C → 0B | 1D
D → 0A | 1C

2) Substitute B in A and C
A → ε | 1(1A | 0C) | 0(0A | 1C)
C → 0(1A | 0C) | 1(0A | 1C)

3) Put C in form C = α | βC
A → ε | 1(1A | 0C) | 0(0A | 1C)
C → 01A | 10A | (00 | 11)C

4) Solve C: C = (00 | 11) *(01A | 10A)

5) Factorize C in A
A → ε | 11A | 00A | (10 | 01)C

6) Substitute C in A
A → ε | 11A | 00A | (10 | 01) (00 | 11) *(01A | 10A)

7) Put A in form A = α | βA
A → ε | (11 | 00 | (10 | 01) (00 | 11) *(01 | 10))A

8) Solve A: A = (11 | 00 | (10 | 01) (00 | 11) *(01 | 10)) *
The other solution: (00 | 11) *((01 | 10)(00 | 11) *(01 | 10)(00 | 11) *) *
Regular expressions, Automata, and all that...

- Regular Expressions
- Deterministic Finite Automata
- Non-Deterministic Finite Automata
- Thompson algorithm
- Least fixed-point of function on languages
- Right-linear (Regular) Grammars
- DeterministicFinite Automata
- Easy!
- Subset construction
- Section 3.9 of Dragon Book
- Minimization (Partition/Refinement)
Top-down Parsing
Position of a Parser in the Compiler Model

Source Program → Lexical Analyzer

Lexical error → Symbol Table

Get next token → Parser and rest of front-end

Token, tokenval → Syntax error

Semantic error → Intermediate representation
The syntax of programming languages

• The syntax of a programming language is typically defined by two grammars
  – Lexical grammar
    • Regular, often presented as regular expressions
    • Terminal symbols are characters
    • Defines tokens
  – Syntax grammar
    • Context-free, often presented in Backus-Naur form
    • Terminal symbols are tokens
    • Defines constructs of the language, not expressible with REs
  – Note: there are non-context free syntactict constructs
    • Variables are declared before use $\Rightarrow$ $\{wcw \mid w \in (a \mid b)^*\}$
    • Number of actual/formal parameters $\Rightarrow$ $\{a^n b^m c^n d^m \mid n > 0, m > 0\}$
Towards parsing

• A parser implements a Context-Free grammar as a recognizer of strings
  – It checks that the input string (of tokens) is generated by the syntax grammar
  – Possibly generates the parse tree
  – Reports syntax errors accurately
  – *Invokes semantic actions*
    • *For static semantics checking, e.g. type checking of expressions, functions, etc.*
    • *For syntax-directed translation of the source code to an intermediate representation*
Parse trees and derivations

- A parse tree may correspond to several derivations
- A parse tree has a unique rightmost (leftmost) derivation

\[ P = E \rightarrow E + E \mid \text{id} \]

1. \[ E \rightarrow_{rm} E + E \rightarrow_{rm} E + \text{id} \rightarrow_{rm} \text{id} + \text{id} \]

2. \[ E \rightarrow_{lm} E + E \rightarrow_{lm} \text{id} + E \rightarrow_{lm} \text{id} + \text{id} \]
Parsing algorithms

• *Universal* (any C-F grammar)
  – Cocke-Younger-Kasimi, Earley
  – Based on dynamic programming, $O(n^3)$

• *Top-down* (C-F grammar with restrictions)
  – Recursive descent (predictive parsing)
  – LL (Left-to-right, Leftmost derivation) methods
  – Linear on certain grammars; easier to do manually

• *Bottom-up* (C-F grammar with restrictions)
  – Operator precedence parsing
  – LR (Left-to-right, Rightmost derivation) methods
    • SLR, canonical LR, LALR
  – Linear on certain grammars; typically generated by tools
Top-Down Parsing

• LL methods (Left-to-right, Leftmost derivation) and recursive-descent parsing

Grammar:

\[
E \rightarrow T + T \\
T \rightarrow ( E ) \\
T \rightarrow - E \\
T \rightarrow id
\]

String:

\[
id + id
\]

Leftmost derivation:

\[
E \Rightarrow_{lm} T + T \\
\Rightarrow_{lm} id + T \\
\Rightarrow_{lm} id + id
\]
LL\((k)\) parsing

• Top-down parsing is efficient if the grammar satisfies certain conditions

• Whenever we have to expand a non-terminal, the next \(k\) token should determine the production to use (*lookahead*)

• In this case the grammar is LL\((k)\)

• Most constructs are LL\((1)\), and we will focus on this class of grammars
Left Recursion

• A grammar is left-recursive if there is a non-terminal $A$ such that $A \Rightarrow^+ A\eta$ for some string $\eta$
  – Example of immediate left-recursion:
    \[ A \rightarrow A\alpha \mid A\beta \mid \gamma \mid \delta \]
  – Left recursion can be indirect

• If the grammar is left-recursive, it cannot be LL($k$): a top-down parser loops forever on certain inputs

• Immediate left recursion elimination:
  \[ A \rightarrow \gamma A_R \mid \delta A_R \quad A_R \rightarrow \alpha A_R \mid \beta A_R \mid \varepsilon \]
A General Left Recursion Elimination Method

• \textit{Input: Grammar G with no cycles or \(\varepsilon\)-productions}
• Arrange the nonterminals in some order \(A_1, A_2, \ldots, A_n\)

\textbf{for} \(i = 1, \ldots, n\) \textbf{do}

\quad \textbf{for} \(j = 1, \ldots, i-1\) \textbf{do}

\quad \quad \text{replace each}

\quad \quad \quad A_i \rightarrow A_j \gamma

\quad \quad \text{with}

\quad \quad \quad A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma

\quad \quad \text{where}

\quad \quad \quad A_j \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k

\quad \textbf{enddo}

\quad \text{eliminate the \textit{immediate left recursion} in } A_i

\textbf{enddo}
Example of left-recursion elimination

\[ A \rightarrow B \; C \mid a \]
\[ B \rightarrow C \; A \mid A \; b \]
\[ C \rightarrow A \; B \mid C \; C \mid a \]

Choose arrangement: A, B, C

\[ i = 1: \text{ nothing to do } \]

\[ i = 2, j = 1: B \rightarrow C \; A \mid \boxed{A} \; b \]
\[ \Rightarrow B \rightarrow C \; A \mid \boxed{B} \; C \; b \mid a \; b \]
\[ \Rightarrow_{(\text{imm})} B \rightarrow C \; A \; B_R \mid a \; b \; B_R \]
\[ B_R \rightarrow C \; b \; B_R \mid \varepsilon \]

\[ i = 3, j = 1: C \rightarrow \boxed{A} \; B \mid C \; C \mid a \]
\[ \Rightarrow C \rightarrow B \; C \; B \mid a \; B \mid C \; C \mid a \]

\[ i = 3, j = 2: C \rightarrow \boxed{B} \; C \; B \mid a \; B \mid C \; C \mid a \]
\[ \Rightarrow C \rightarrow C \; A \; B_R \; C \; B \mid a \; b \; B_R \; C \; B \mid a \; B \mid C \; C \mid a \]
\[ \Rightarrow_{(\text{imm})} C \rightarrow a \; b \; B_R \; C \; B \; C_R \mid a \; B \; C_R \mid a \; C_R \]
\[ C_R \rightarrow A \; B_R \; C \; B \; C_R \mid C \; C_R \mid \varepsilon \]
Example of left-recursion elimination:
Grammars for expressions

\[
E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow (E) \mid \text{id}
\]

Grammar after left recursion elimination

\[
E \rightarrow T E' \\
E' \rightarrow + T E' \mid \epsilon \\
T \rightarrow F T' \\
T' \rightarrow * F T' \mid \epsilon \\
F \rightarrow (E) \mid \text{id}
\]
Left Factoring

• If a nonterminal has two or more productions whose right-hand sides start with the same symbol, the grammar is not LL(1)
• Example:
  – stmt ::= if expr then stmt else stmt
  | if expr then stmt
• Solution: replace productions
  \[ A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \ldots | \alpha \beta_n | \gamma \]
  with
  \[ A \rightarrow \alpha A_R | \gamma \]
  \[ A_R \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n \]
• Example:
  – stmt ::= if expr then stmt stmt'
  – stmt' ::= else stmt | \varepsilon
Predictive Parsing

• Eliminate left recursion from grammar
• Left factor the grammar
• Compute FIRST and FOLLOW, and check if the grammar is LL(1)
• FIRST and FOLLOW are used in the parsing algorithm
• Two variants:
  – Recursive (recursive-descent parsing)
  – Non-recursive (table-driven parsing)
FIRST (Revisited)

- \( \text{FIRST}(\alpha) = \{ \text{the set of terminals that begin all strings derived from } \alpha \} \)
- \( \text{FIRST}(a) = \{a\} \) if \( a \in T \)
- \( \text{FIRST}(\varepsilon) = \{\varepsilon\} \)
- \( \text{FIRST}(A) = \bigcup_{A \rightarrow \alpha} \text{FIRST}(\alpha) \) for \( A \rightarrow \alpha \in P \)
- \( \text{FIRST}(X_1X_2...X_k) = \)
  - \( \text{if for all } j = 1, ..., i-1 : \varepsilon \in \text{FIRST}(X_j) \text{ then} \)
    - add non-\( \varepsilon \) in \( \text{FIRST}(X_i) \) to \( \text{FIRST}(X_1X_2...X_k) \)
  - \( \text{if for all } j = 1, ..., k : \varepsilon \in \text{FIRST}(X_j) \text{ then} \)
    - add \( \varepsilon \) to \( \text{FIRST}(X_1X_2...X_k) \)
FOLLOW

- \( \text{FOLLOW}(A) = \{ \text{the set of terminals that can immediately follow nonterminal } A \} \)

- \( \text{FOLLOW}(A) = \)
  
  \textbf{for all } (B \rightarrow \alpha A \beta) \in \mathcal{P} \textbf{ do}
  
  add FIRST(\beta) \setminus \{\varepsilon\} to FOLLOW(A)
  
  \textbf{for all } (B \rightarrow \alpha A \beta) \in \mathcal{P} \text{ and } \varepsilon \in \text{FIRST}(\beta) \textbf{ do}
  
  add FOLLOW(B) to FOLLOW(A)
  
  \textbf{for all } (B \rightarrow \alpha A) \in \mathcal{P} \textbf{ do}
  
  add FOLLOW(B) to FOLLOW(A)
  
  \textbf{if} A \text{ is the start symbol } S \textbf{ then}
  
  add $ to FOLLOW(A)
LL(1) Grammar

- A grammar $G$ is LL(1) if it is not left recursive and for each collection of productions
  \[ A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n \]
  for nonterminal $A$ the following holds:

  1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ for all $i \neq j$
  2. if $\alpha_i \Rightarrow^* \varepsilon$ then
     2.a. $\alpha_j \not\Rightarrow^* \varepsilon$ for all $i \neq j$
     2.b. $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ for all $i \neq j$
Non-LL(1) Examples

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Not LL(1) because:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow S\ a \mid a$</td>
<td>Left recursive</td>
</tr>
<tr>
<td>$S \rightarrow a\ S \mid a$</td>
<td>FIRST(a S) $\cap$ FIRST(a) $\neq$ $\emptyset$</td>
</tr>
<tr>
<td>$S \rightarrow a\ R \mid \varepsilon$</td>
<td>For $R$: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$</td>
</tr>
<tr>
<td>$R \rightarrow S \mid \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow a\ R\ a$</td>
<td>For $R$:</td>
</tr>
<tr>
<td>$R \rightarrow S \mid \varepsilon$</td>
<td>FIRST(S) $\cap$ FOLLOW(R) $\neq$ $\emptyset$</td>
</tr>
</tbody>
</table>
Recursive-Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal’s syntactic category of input tokens
- When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information
Using FIRST and FOLLOW in a Recursive-Descent Parser

\[
\begin{align*}
expr & \rightarrow \text{term rest} \\
rest & \rightarrow + \text{term rest} \\
& \mid - \text{term rest} \\
& \mid \varepsilon \\
term & \rightarrow \text{id}
\end{align*}
\]

\begin{verbatim}
procedure rest();
begin
  if lookahead in FIRST(+ term rest) then
    match('+'); term(); rest()
  else if lookahead in FIRST(- term rest) then
    match('-'); term(); rest()
  else if lookahead in FOLLOW(rest) then
    return
  else error()
end;
\end{verbatim}

where \( \text{FIRST}(+ \text{term rest}) = \{ + \} \)
\( \text{FIRST}(- \text{term rest}) = \{ - \} \)
\( \text{FOLLOW}(rest) = \{ $ \} \)
Non-Recursive Predictive Parsing: Table-Driven Parsing

- Given an LL(1) grammar $G = (N, T, P, S)$ construct a table $M$ and use a driver program with a stack.
- The stack replaces the runtime stack of the recursive algorithm. It will contain symbols of the grammar.

```
input: a + b $
```

```
stack: X Y Z $
```

```
Predictive parsing program (driver)
```

```
Parsing table $M$
```

output: X Y Z $

Constructing an LL(1) Predictive Parsing Table

- Table $M$ has one entry $M[A, a]$ for each $A \in N$ and $a \in T$
- Entry $M[A, a]$ contains the production to apply when $A$ has to be reduced and $a$ is the lookahead

```plaintext
for each production $A \rightarrow \alpha$ do
    for each $a \in \text{FIRST}(\alpha)$ do
        add production $A \rightarrow \alpha$ to $M[A,a]$
    enddo
if $\epsilon \in \text{FIRST}(\alpha)$ then
    for each $b \in \text{FOLLOW}(A)$ do
        add $A \rightarrow \alpha$ to $M[A,b]$
    enddo
endif
enddo
```

- Mark each undefined entry in $M$ error
- **Note:** The grammar is LL(1) iff $M[A, a]$ contains at most one production for each $A \in N$ and $a \in T$
Example Table

\[
\begin{align*}
E & \rightarrow TE_R \\
E_R & \rightarrow + TE_R | \varepsilon \\
T & \rightarrow FT_R \\
T_R & \rightarrow * FT_R | \varepsilon \\
F & \rightarrow (E) | \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>$A \rightarrow \alpha$</th>
<th>FIRST($\alpha$)</th>
<th>FOLLOW($A$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow TE_R$</td>
<td>(id)</td>
<td>$$)</td>
</tr>
<tr>
<td>$E_R \rightarrow + TE_R$</td>
<td>+</td>
<td>$$)</td>
</tr>
<tr>
<td>$E_R \rightarrow \varepsilon$</td>
<td>$\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow FT_R$</td>
<td>(id)</td>
<td>+$$)</td>
</tr>
<tr>
<td>$T_R \rightarrow * FT_R$</td>
<td>*</td>
<td>+$$)</td>
</tr>
<tr>
<td>$T_R \rightarrow \varepsilon$</td>
<td>$\varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>()</td>
<td>*+$$)</td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>id</td>
<td>*+$$)</td>
</tr>
</tbody>
</table>

<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 \rightarrow TE_R$</td>
<td></td>
<td></td>
<td>$E \rightarrow TE_R$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E_R$</td>
<td>$E_R \rightarrow + TE_R$</td>
<td></td>
<td></td>
<td>$E_R \rightarrow \varepsilon$</td>
<td>$E_R \rightarrow \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow FT_R$</td>
<td></td>
<td></td>
<td>$T \rightarrow FT_R$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_R$</td>
<td>$T_R \rightarrow \varepsilon$</td>
<td>$T_R \rightarrow * FT_R$</td>
<td></td>
<td>$T_R \rightarrow \varepsilon$</td>
<td>$T_R \rightarrow \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow \text{id}$</td>
<td></td>
<td></td>
<td>$F \rightarrow (E)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Predictive Parsing Program (Driver)

push($)
push(S)
\(a := \text{lookahead}\)
repeat
\(X := \text{pop()}\)
\(\text{if } X \text{ is a terminal or } X = $ \text{ then}\)
\(\quad \text{match}(X) \quad // \text{moves to next token and } a := \text{lookahead}\)
\(\text{else if } M[X,a] = X \rightarrow Y_1Y_2...Y_k \text{ then}\)
\(\quad \text{push}(Y_k, Y_{k-1}, ..., Y_2, Y_1) \quad // \text{such that } Y_1 \text{ is on top}\)
\(\quad \ldots \text{ invoke actions and/or produce IR output } \ldots\)
\(\text{else} \quad \text{error()}\)
endif
until \(X = $\)
## Example Table-Driven Parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Production applied</th>
<th>Prod. applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id*id$</td>
<td>$E \rightarrow T E_R$</td>
<td>$T_R \rightarrow * F T_R$</td>
</tr>
<tr>
<td>$E_R T_R$</td>
<td>id+id*id$</td>
<td>$T \rightarrow F T_R$</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$E_R T_R F_R$</td>
<td>id+id*id$</td>
<td>$F \rightarrow id$</td>
<td>$T_R \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E_R T_R$</td>
<td>+id*id$</td>
<td>$T_R \rightarrow \epsilon$</td>
<td>$E_R \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E_R$</td>
<td>+id*id$</td>
<td>$E_R \rightarrow + T E_R$</td>
<td>$E_R \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E_R T_R$</td>
<td>id*id$</td>
<td>$T \rightarrow F T_R$</td>
<td>$T_R \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E_R T_R F_R$</td>
<td>$E_R \rightarrow \epsilon$</td>
<td>$E_R \rightarrow \epsilon$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Production applied</th>
<th>E R T R id</th>
<th>id*id$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prod. applied</td>
<td>$T_R \rightarrow * F T_R$</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td></td>
<td>$T_R \rightarrow \epsilon$</td>
<td>$E_R \rightarrow \epsilon$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Prod. applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$E \rightarrow T E_R$</td>
<td>$E \rightarrow T E_R$</td>
</tr>
<tr>
<td>$E_R$</td>
<td>$E_R \rightarrow + T E_R$</td>
<td>$E_R \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow F T_R$</td>
<td>$T \rightarrow F T_R$</td>
</tr>
<tr>
<td>$T_R$</td>
<td>$T_R \rightarrow \epsilon$</td>
<td>$T_R \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow (E)$</td>
</tr>
</tbody>
</table>
LL(1) Grammars are Unambiguous

Ambiguous grammar

\[ S \rightarrow i E t S S_R | a \]
\[ S_R \rightarrow e S | \varepsilon \]
\[ E \rightarrow b \]

Error: duplicate table entry

<table>
<thead>
<tr>
<th></th>
<th>( A \rightarrow \alpha )</th>
<th>FIRST(( \alpha ))</th>
<th>FOLLOW(( A ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow i E t S S_R )</td>
<td>( i )</td>
<td>( e ) $</td>
<td></td>
</tr>
<tr>
<td>( S \rightarrow a )</td>
<td>( a )</td>
<td>( e )$</td>
<td></td>
</tr>
<tr>
<td>( S_R \rightarrow e S )</td>
<td>( e )</td>
<td>( e )$</td>
<td></td>
</tr>
<tr>
<td>( S_R \rightarrow \varepsilon )</td>
<td>( \varepsilon )</td>
<td>( \varepsilon ) $</td>
<td></td>
</tr>
<tr>
<td>( E \rightarrow b )</td>
<td>( b )</td>
<td>( t )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>i</th>
<th>t</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( S \rightarrow a )</td>
<td></td>
<td></td>
<td>( S \rightarrow i E t S S_R )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S_R )</td>
<td></td>
<td></td>
<td>( S_R \rightarrow \varepsilon )</td>
<td>( S_R \rightarrow \varepsilon )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( S_R \rightarrow e S )</td>
<td>( S_R \rightarrow e S )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( E )</td>
<td>( E \rightarrow b )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Error Handling

• A good compiler should assist in identifying and locating errors
  
  – *Lexical errors*: compiler can easily recover and continue (e.g. misspelled identifiers)
  – *Syntax errors*: can almost always recover (e.g. missing ‘;’ or ‘{‘, misplaced *case*)
  – *Static semantic errors*: can sometimes recover (e.g. type mismatches, variable used before declaration)
  – *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required (e.g. null pointer, division by zero, invalid array access)
  – *Logical errors*: hard or impossible to detect (e.g. if (b = true) ... )
Viable-Prefix Property

• The *viable-prefix property* of parsers allows early detection of syntax errors
  – Enjoyed by LL(1), LR(1) parsers
  – Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  – How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

\[
\text{Prefix} \begin{cases} 
\text{... for (;) ...} \\
\text{Error is detected here}
\end{cases}
\]
Error Recovery Strategies

• *Panic mode*
  – Discard input until a token in a set of designated “synchronizing tokens” is found (e.g. “}”, “;”)

• *Phrase-level recovery*
  – Perform local correction on the input to repair the error

• *Error productions*
  – Augment grammar with productions for erroneous constructs

• *Global correction*
  – Choose a minimal sequence of changes to obtain a global least-cost correction
Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

<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 \rightarrow T E_R$</td>
<td>$E \rightarrow + T E_R$</td>
<td>$E \rightarrow T E_R$</td>
<td>synch</td>
<td>synch</td>
<td></td>
</tr>
<tr>
<td>$E_R$</td>
<td>$E_R \rightarrow + T E_R$</td>
<td></td>
<td>$E_R \rightarrow \epsilon$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow F T_R$</td>
<td>synch</td>
<td>$T \rightarrow F T_R$</td>
<td>synch</td>
<td>synch</td>
<td></td>
</tr>
<tr>
<td>$T_R$</td>
<td></td>
<td>$T_R \rightarrow \epsilon$</td>
<td>$T_R \rightarrow * F T_R$</td>
<td>$T_R \rightarrow \epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow \text{id}$</td>
<td>synch</td>
<td>synch</td>
<td>$F \rightarrow ( E )$</td>
<td>synch</td>
<td>synch</td>
</tr>
</tbody>
</table>

$\text{synch}$: the driver pops current nonterminal $A$ and skips input till synch token or skips input until one of FIRST($A$) is found.

**FOLLOW**

- $\text{FOLLOW}(E) = \{ ) \; $ \} 
- $\text{FOLLOW}(E_R) = \{ ) \; $ \} 
- $\text{FOLLOW}(T) = \{ + ) \; $ \} 
- $\text{FOLLOW}(T_R) = \{ + ) \; $ \} 
- $\text{FOLLOW}(F) = \{ + * ) \; $ \} 

**Pro:** Can be automated

**Cons:** Error messages are needed
Phrase-Level Recovery

Change input stream by inserting missing tokens
For example: \textbf{id id} is changed into \textbf{id * id}

Pro: Can be fully automated
Cons: Recovery not always intuitive

<table>
<thead>
<tr>
<th>(\text{id})</th>
<th>+</th>
<th>*</th>
<th>( )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E)</td>
<td>(E \rightarrow TE_R)</td>
<td>(E \rightarrow TE_R)</td>
<td>synch</td>
<td>synch</td>
</tr>
<tr>
<td>(E_R)</td>
<td>(E_R \rightarrow + TE_R)</td>
<td>(E_R \rightarrow \epsilon)</td>
<td>(E_R \rightarrow \epsilon)</td>
<td></td>
</tr>
<tr>
<td>(T)</td>
<td>(T \rightarrow FT_R)</td>
<td>(T \rightarrow FT_R)</td>
<td>synch</td>
<td>synch</td>
</tr>
<tr>
<td>(T_R)</td>
<td>(\text{insert} *)</td>
<td>(T_R \rightarrow \epsilon)</td>
<td>(T_R \rightarrow \epsilon)</td>
<td></td>
</tr>
<tr>
<td>(F)</td>
<td>(F \rightarrow \text{id})</td>
<td>(F \rightarrow (E))</td>
<td>synch</td>
<td>synch</td>
</tr>
</tbody>
</table>

\textit{insert} *: driver inserts missing * and retries the production
## Error Productions

Add "error production":

\[ T_R \rightarrow F T_R \]

to ignore missing *, e.g.: \texttt{id id}

### Pro: Powerful recovery method

### Cons: Manual addition of productions

<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 \rightarrow T E_R )</td>
<td>( E \rightarrow T E_R )</td>
<td>( E \rightarrow T E_R )</td>
<td>synch</td>
<td>synch</td>
<td></td>
</tr>
<tr>
<td>E_R</td>
<td>( E_R \rightarrow + T E_R</td>
<td>\epsilon )</td>
<td>( E_R \rightarrow + T E_R )</td>
<td>( E_R \rightarrow \epsilon )</td>
<td>( E_R \rightarrow \epsilon )</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>( T \rightarrow F T_R )</td>
<td>synch</td>
<td>( T \rightarrow F T_R )</td>
<td>synch</td>
<td>synch</td>
<td></td>
</tr>
<tr>
<td>T_R</td>
<td>( T_R \rightarrow F T_R )</td>
<td>( T_R \rightarrow \epsilon )</td>
<td>( T_R \rightarrow * F T_R )</td>
<td>( T_R \rightarrow \epsilon )</td>
<td>( T_R \rightarrow \epsilon )</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>( F \rightarrow \text{id} )</td>
<td>synch</td>
<td>synch</td>
<td>( F \rightarrow ( E ) )</td>
<td>synch</td>
<td>synch</td>
</tr>
</tbody>
</table>
Shift-Reduce Parsing

Grammar:

\[
S \rightarrow a \, A \, B \, e \\
A \rightarrow A \, b \, c \mid b \\
B \rightarrow d
\]

Reducing a sentence:

\[
a \, b \, b \, c \, d \, e \\
a \, A \, b \, c \, d \, e \\
a \, A \, d \, e \\
a \, A \, B \, d \, e \\
S
\]

Shift-reduce corresponds to a rightmost derivation:

\[
S \Rightarrow_{rm} a \, A \, B \, e \\
\Rightarrow_{rm} a \, A \, d \, e \\
\Rightarrow_{rm} a \, A \, b \, c \, d \, e \\
\Rightarrow_{rm} a \, b \, b \, c \, d \, e
\]

Diagram: 

```
A   A   A   A   A
|   |   |   |   |
\_b_\_b_\_c_\_d_\_e
```

```
A
|   |   |   |
\_b_\_b_\_c_\_d_\_e
```

```
A
|   |   |   |
\_b_\_b_\_c_\_d_\_e
```

```
S
```

```
A
|   |
\_b_\_b_\_c_\_d_\_e
```

```
A
|   |
\_b_\_b_\_c_\_d_\_e
```

```
A
|   |
\_b_\_b_\_c_\_d_\_e
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```

```
A
```

```
B
```