Parsing

Matthew Dwyer
324E Nichols Hall
dwyer@cis.ksu.edu
http://www.cis.ksu.edu/~dwyer

Compiler Architecture

SCAN → PARSE → WEED

RESOURCE → TYPE

CODE → OPTIMIZE

SYMBOL

EMIT
The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program
  - parse tree is generated if the input is a legal program
  - if input is an illegal program, syntax errors are issued
- **Note:** Instead of parse tree, some parsers produce:
  - abstract syntax tree (AST) + symbol table
- In the following, we'll assume that parse tree is generated.
Comparison with Scanner

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanner</td>
<td>String of characters</td>
<td>String of tokens</td>
</tr>
<tr>
<td>Parser</td>
<td>String of tokens</td>
<td>Parse tree</td>
</tr>
</tbody>
</table>

Example

- The program:
  \[ x \times y + z \]

- Input to parser:
  ID TIMES ID PLUS ID
  We'll write tokens as follows:
  \[ id \times id + id \]

- Output of parser:
  The parse tree →
Can we use REs?

Write an automaton that accepts strings
\{“a”, “(a)”, “((a))”, “(((a)))”\}

\{“a”, “(a)”, “((a))”, “(((a)))”, …“(ka)k” \}
where \(a^k\) means \(a\ a\ a\ \ldots\ a\)

k times

Can we use REs?

What programs are generated by?
\texttt{digit+ ( (“+” | “-” | “*” | “/” ) digit+ )*}

What important properties does this RE fail to express?
Parser: Overview

Parser plays two important roles

**Recognizer**: not all strings of tokens are programs
   - must distinguish between valid and invalid strings of tokens

**Translator**: must expose program structure
   - e.g., associativity and precedence
   - hence must return the parse tree

Similar requirements as for scanner, we need:

A language for describing valid strings of tokens
   - To function like REs in scanner
   - Context-free Grammars (CFG)

A method for distinguishing valid from invalid strings of tokens
   - To function like the DFA and algorithm in scanner
   - Push-down Automata (PDA)
Context-free Grammar

In English:
- An integer is an arithmetic expression.
- If $\text{exp}_1$ and $\text{exp}_2$ are arithmetic expressions, then so are the following:
  - $\text{exp}_1 - \text{exp}_2$
  - $\text{exp}_1 / \text{exp}_2$
  - $( \text{exp}_2 )$

As a CFG:
- $E \rightarrow \text{intlit}$
- $E \rightarrow E - E$
- $E \rightarrow E / E$
- $E \rightarrow ( E )$

Reading the CFG

- The grammar has five terminal symbols: $\text{intlit}, -, /, (, )$
  - terminals are tokens returned by the scanner.
- The grammar has one non-terminal symbol: $E$
  - non-terminals describe valid (sub)sequences of tokens
- The grammar has four productions or rules
  - each of the form: $E \rightarrow \alpha$
  - left-hand side is a single non-terminal
  - right-hand side ($\alpha$) is either
    - a sequence of one or more terminals and/or non-terminals
    - $\epsilon$ (an empty production)
Example, revisited

A more compact way to write previous grammar:

\[ E \rightarrow \text{intlit} \mid E - E \mid E / E \mid (E) \]

or

\[ E \rightarrow \text{intlit} \mid E - E \mid E / E \mid (E) \]

A formal definition of CFGs

- A CFG consists of
  - A set of terminals \( T \)
  - A set of non-terminals \( N \)
  - A start symbol \( S \) (a non-terminal)
  - A set of productions:
    \[ X \rightarrow Y_1 Y_2 \cdots Y_n \]
    where \( X \in N \) and \( Y_i \in T \cup N \cup \{\varepsilon\} \)
Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

The Language of a CFG

The language defined by a CFG is the set of terminal strings that can be derived from the start symbol of the grammar.

**Derivation:** Read productions as rules:

\[ X \rightarrow Y_1 \cdots Y_n \]

Means \( X \) can be replaced by \( Y_1 \cdots Y_n \)
Derivation

A sequence of grammar rule applications leading from the start symbol

Basic Idea

1. Begin with a string consisting of the start symbol “S”
2. Replace any non-terminal \( X \) in the string by the right-hand side of some production
   \[
   X \rightarrow Y_1 \cdots Y_n
   \]
3. Repeat (2) until there are no non-terminals in the string

Derivation: an example

CFG:

- \( E \rightarrow \text{id} \)
- \( E \rightarrow E + E \)
- \( E \rightarrow E \ast E \)
- \( E \rightarrow (E) \)

Is string \( \text{id} \ast \text{id} + \text{id} \) in the language defined by the grammar?

 derivation:

- \( E \rightarrow E \rightarrow E+E \rightarrow E \ast E+E \rightarrow \text{id} \ast E+E \rightarrow \text{id} \ast \text{id} + E \rightarrow \text{id} \ast \text{id} + \text{id} \)
More formally, write

\[ X_1 \cdots X_i \cdots X_n \rightarrow X_1 \cdots X_{i-1} Y_1 \cdots Y_m X_{i+1} \cdots X_n \]

if there is a production

\[ X_i \rightarrow Y_1 \cdots Y_m \]

A *sentential form* is a sequence of terminals and non-terminals

The Language of a CFG (Cont.)

Write

\[ X_1 \cdots X_n \rightarrow^* Y_1 \cdots Y_m \]

if

\[ X_1 \cdots X_n \rightarrow \cdots \rightarrow Y_1 \cdots Y_m \]

in 0 or more steps
The Language of a CFG

Let $G$ be a context-free grammar with start symbol $S$. Then the language of $G$ is:

$$\left\{a_1 \ldots a_n \mid S \Rightarrow a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

Examples

Strings of balanced parentheses $\{(')^i \mid i \geq 0\}$

The grammar:

$$\begin{align*}
S & \rightarrow (S) \\
S & \rightarrow \epsilon \\
S & \rightarrow (S) \quad \text{same as} \\
& \mid \epsilon
\end{align*}$$
Arithmetic Example

Simple arithmetic expressions:
\[ E \rightarrow E + E \mid E \times E \mid (E) \mid id \]

Some elements of the language:
- id
- id + id
- (id) id
- id * id
- (id) * id
- id * (id)

Notes on CFGs

Membership in a language is “yes” or “no”
– we also need parse tree of the input!
– furthermore, we must handle errors gracefully

Need an “implementation” of CFG’s,
– i.e. the parser

Form of the grammar is important for generating a parser
A derivation is a sequence of productions

\[ S \rightarrow \cdots \rightarrow \cdots \rightarrow \cdots \]

A derivation can be drawn as a tree

– Start symbol is the tree’s root
– For a production \( X \rightarrow Y_1 \cdots Y_n \) add children \( Y_1 \cdots Y_n \) to node \( X \)

### Derivation Example

- Grammar

\[
E \rightarrow E+E \mid E*E \mid (E) \mid id
\]

- String

\[
id \ast id + id
\]
Derivation Example (Cont.)

\[
E 
\rightarrow E + E 
\]

Derivation Example (Cont.)

\[
E 
\rightarrow E + E 
\rightarrow E * E + E 
\]
Derivation Example (Cont.)

E
→ E+E
→ E * E+E
→ id * E + E

Derivation Example (Cont.)

E
→ E+E
→ E * E+E
→ id * E + E
→ id * id + E
Derivation Example (Cont.)

E
→ E+E
→ E*E+E
→ id*E + E
→ id*id + E
→ id*id + id

Derivation Example (Cont.)

E
→ E+E
→ E*E+E
→ id*E + E
→ id*id + E
→ id*id + id
**Notes on Derivations**

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

---

**Left-most and Right-most Derivations**

- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation
Derivation Example (Cont.)

E
→ E+E

Derivation Example (Cont.)

E
→ E+E
→ E+id
Derivation Example (Cont.)

E
→ E+E
→ E+id
→ E*E + id

CIS 706 Translators I
Derivation Example (Cont.)

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

Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree

- The difference is the order in which branches are added
Summary of Derivations

• We are not just interested in whether $s \in L(G)$
  – we need a parse tree for $s$
• A derivation defines a parse tree
  – but one parse tree may have many derivations
• Left-most and right-most derivations are important in parser implementation

Multiple Parse Trees

The string “id * id + id” has two parse trees

```
  E
  /|
 /  |
| E + E|
| |   |
| E * E id|
| id    id|
```
```
  E
  /|
 /  |
| E * E|
| id   |
|   E + E|
| id    id|
```
For You To Do

for each of the two parse trees, find the corresponding **left-most derivation**

for each of the two parse trees, find the corresponding **right-most derivation**

---

**Ambiguity**

- A grammar is *ambiguous* if for some string (the following three conditions are equivalent)
  - it has more than one parse tree
  - if there is more than one right-most derivation
  - if there is more than one left-most derivation

- Ambiguity is **BAD**
  - Want operator precedence enforced!
Resolving Ambiguity

- Rewrite the grammar
  - use a different nonterminal for each precedence level
  - start with the lowest precedence (MINUS)

\[
E \rightarrow E - E \mid E / E \mid (E) \mid t\delta
\]

rewrite to

\[
E \rightarrow E - E \mid T
T \rightarrow T / T \mid \Phi
\Phi \rightarrow t\delta \mid (E)
\]

For You To Do

Attempt to construct a parse tree for
“id − id / id”
that shows the wrong precedence.
Associativity

• The grammar captures operator precedence, but it is still does not capture the proper meaning

• Fails to express that both subtraction and division are *left* associative;
  
  \[ 5-3-2 = ((5-3)-2) \text{ and not } (5-(3-2)) \]

For You To Do

Draw two parse trees for the expression

\[ 5-3-2 \]

using the grammar given above;

– one that correctly groups 5-3
– one that incorrectly groups 3-2
Recursion

A grammar is *recursive* in nonterminal $X$ if $X \Rightarrow^{*} \ldots X \ldots$
– in one or more steps, $X$ derives a sequence of symbols that includes an $X$

A grammar is *left recursive* in $X$ if $X \Rightarrow^{*} X \ldots$
– in one or more steps, $X$ derives a sequence of symbols that *starts* with an $X$

A grammar is *right recursive* in $X$ if $X \Rightarrow^{*} \ldots X$
– in one or more steps, $X$ derives a sequence of symbols that *ends* with an $X$

Resolving Associativity

• The grammar given above is both left and right recursive in nonterminals $E$ and $T$
• To correctly expresses operator associativity:
  – For left associativity, use left recursion.
  – For right associativity, use right recursion.
• Here's the correct grammar:
  
  $E \rightarrow E - T | T$
  $T \rightarrow T / F | F$
  $F \rightarrow \text{id} | (E)$
Ambiguity: The Dangling Else

Consider the grammar

\[ E \rightarrow \text{if } E \text{ then } E \]
| \[ \text{if } E \text{ then } E \text{ else } E \]
| \[ \text{print} \]

This grammar is also ambiguous.

The Dangling Else: Example

The expression

\[ \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \]

has two parse trees:

\[ \text{if } E_1 \text{ if } E_2 \text{ E_3 E_4} \]
\[ \text{if } E_1 \text{ if } E_2 \text{ E_3 E_4} \]
The Dangling Else: A Fix

else matches the closest unmatched then

We can describe this in the grammar

\[
E \rightarrow \text{MIF} \quad /\!* \text{all then are matched } */\!
\quad | \quad \text{UIF} \quad /\!* \text{some then are unmatched } */\!
\]

\[
\text{MIF} \rightarrow \text{if } E \text{ then MIF else MIF}
\quad | \quad \text{print}
\]

\[
\text{UIF} \rightarrow \text{if } E \text{ then E}
\quad | \quad \text{if } E \text{ then MIF else UIF}
\]

The Dangling Else: Example Revisited

The expression if \( E_1 \) then if \( E_2 \) then \( E_3 \) else \( E_4 \)

- A valid parse tree (for a UIF)
- Not valid because the then expression is not a MIF
Precedence and Associativity Declarations

• Instead of rewriting the grammar
  – Use the more natural (ambiguous) grammar
  – Along with disambiguating declarations

• Most parser generators allow precedence and associativity declarations to disambiguate grammars