Reading

- Chapter 3
  - Introduction
  - Section 3.1

Parsing

- = determining whether a string of tokens can be generated by a grammar
- Two classes based on order in which parse tree is constructed:
  - Top-down parsing
    - Start construction at root of parse tree
  - Bottom-up parsing
    - Start at leaves and proceed to root
Derivations and Parse Trees

A *derivation* is a sequence of productions

A derivation can be drawn as a tree

- Start symbol is the tree’s root
- For a production $N \rightarrow X_1 \ldots X_n$, add $X_1 \ldots X_n$ as children to node $N$

Derivation Example

- $S \rightarrow E + E | E \cdot E$
- $E \rightarrow id | (E)$
- Derivation for: $id \cdot id + id$
- Derivation for: $id + id + id$
- See board

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
Terminals

- Terminals are called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals are the tokens of the language represented by the grammar

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

Right-most Derivation in Detail (1)

- E
- E E
- E E
- E + E
- E + id
- E * E + id
- E * id + id
- id * id + id
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

Parse trees

Question 1:
- for each of the two parse trees, find the corresponding left-most derivation

Question 2:
- for each of the two parse trees, find the corresponding right-most derivation
Ambiguity (Cont.)

- 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
  - Leaves meaning of some programs ill-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity
  - Most direct method is to rewrite grammar unambiguously
    - Enforces precedence of * over +
  - Separate (external of grammar) conflict-resolution rules
    - E.g., precedence or associativity rules (e.g. via parser tool declarations)
    - Same idea we saw with scanning: instead of complicated RE’s use symbol table to recognize keywords

Expression Grammars (precedence)

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

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

  rewrite to

  \[ E \rightarrow E - E \mid T \]
  \[ T \rightarrow T / T \mid F \]
  \[ F \rightarrow \text{id} \mid (E) \]
Example

parse tree for $id - id / id$

\[
E \to E - E \mid T
\]

\[
T \to T / T \mid F
\]

\[
F \to id \mid (E)
\]

More than one parse tree?

- Attempt to construct a parse tree for $id - id / id$ that shows the wrong precedence.

Question:

- Why do you fail to construct this parse tree?

Associativity

- The grammar captures operator precedence, but it is still ambiguous!
  - fails to express that both subtraction and division are left associative;
    - e.g., $5-3-2$ is equivalent to: $((5-3)-2)$ and not to: $(5-(3-2))$. 
Recursion

- A grammar is recursive in nonterminal X if:
  - X →+ … X ...
  - recall that →+ means "in one or more steps, X derives a sequence of symbols that includes an X"
- A grammar is left recursive in X if:
  - X →+ X ...
  - 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 →+ … X
  - in one or more steps, X derives a sequence of symbols that ends with an X

How to fix associativity

- The grammar given above is both left and right recursive in nonterminals exp and term
- try at home: write the derivation steps that show this.
- To correctly express operator associativity:
  - For left associativity, use left recursion.
  - For right associativity, use right recursion.
- Here's the correct grammar:

```
E → E - T | T
T → T / F | F
F → id | ( E )
```

Ambiguity - has both left and right associativity

Old grammar - corrected for precedence

```
E → E - E | T
T → T / T | F
F → id | ( E )
```

Here's the correct grammar - corrected for both precedence and associativity

```
E → E - T | T
T → T / F | F
F → id | ( E )
```
Ambiguity: More than operator - 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

- Typically we want the second form

The Dangling Else: A Fix

- else matches the closest unmatched then
- We can describe this in the grammar
  \[ S \rightarrow \text{WithElse}/* all then are matched */ \]
  \[ | \text{LastElse}/* some then are unmatched */ \]
  \[ \text{WithElse} \rightarrow \text{if } E \text{ then } \text{WithElse} \text{ else } \text{WithElse} \]
  \[ | \text{print} \]
  \[ \text{LastElse} \rightarrow \text{if } E \text{ then } S \]
  \[ | \text{if } E \text{ then } \text{WithElse} \text{ else } \text{LastElse} \]
- Describes the same set of strings
The Dangling Else: Example Revisited

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

- Examples …

Associativity Declarations

- Consider the grammar
  \[ E \rightarrow E - E \mid \text{int} \]
- Ambiguous: two parse trees of \( \text{int} - \text{int} - \text{int} \)

- Left associativity declaration: \%left +
Precedence Declarations

- Consider the grammar

\[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]

- And the string int + int * int

- Precedence declarations:
  \%left +
  \%left *

Parenthetical Excursion into the Land of Grammars

- Just FYI - if you are theoretically inclined you may want to know this
- Relevant for the practical compiler hacker wrt. kinds of constructs can be put into a CFG

Chomsky Language Classification based on Grammar

- 4 types of grammars
  - Type 0 grammar - unrestricted or recursive
    - Form of rule:
      \[ \alpha \rightarrow \beta \mid \alpha \in (N \cup T)^* \mid \beta \in (N \cup T)^* \]
      - no restrictions on form of grammar rules
    - Example: \( aAB \rightarrow ab \)
      \[ A \rightarrow \varepsilon \] can have empty productions
      \[ | \alpha | \mid | \beta | \]
Type 1 grammar - context sensitive
Form of rule:
\[ a \, A \, \beta \to \alpha \, \gamma \, \beta, \text{ where } A \in N' \]
\[ \gamma \in (N \cup T)^* \]

| A | \leq | \gamma |

Replace A by \gamma only if found in context of \alpha and \beta.
No erase rules

Type 2 grammar - context free
\[ A \to \alpha, \ A \in N, \ \alpha \in (N \cup T)^* \]

Programming language constructs - mostly context free
but not all -
- declarations must proceed use
- number of formals and actuals agree
  must check this in semantic phase

Type 3 grammar - regular
rules: \[ A \to a \]
or \[ A \to aB, \ a \in T, \ A, B \in N \]
tokens defined by type 3 grammar
So lexical analysis - regular expressions - can only describe regular languages
End of Excursion

- Time to wake up again

Parsing Algorithms

- Can parse any grammar
  - Earley’s algorithm - expensive $O(n^3)$
- Top-down
  - Start from $S$ and build input string
  - Leftmost derivation
  - Can be easily written by hand
    - Recursive descent
    - Alternative implementation
    - Predictive parsing
  - Only works for some grammars (LL(k))

Bottom-up Parsing

- Reduce input string to $S$ (root of tree)
- Rightmost derivation starting from leaves
- Shift-reduce: shift symbols on a stack, reduce to LHS of production when a “handle” is identified
- LR(k) parser
  - Hard to build by hand
  - Tools
  - More grammars can be parsed

12
Intro to Top-Down Parsing

- The parse tree is constructed
  - From the top
  - From left to right
- Terminals are seen in order of appearance in the token stream:

Recursive Descent Parsing

- Consider the grammar:
  \[ E \rightarrow T + E | T \]
  \[ T \rightarrow \text{int} | \text{int} * T | ( E ) \]

- Token stream is: \text{int}_5 * \text{int}_2
- Start with top-level non-terminal \( E \)
- Try the rules for \( E \) in order

Recursive Descent Parsing. Example (Cont.)

- Try \( E_0 \rightarrow T_1 + E_2 \)
- Then try a rule for \( T_1 \rightarrow ( E_3 ) \)
  - But ( does not match input token \( \text{int}_3 \)
- Try \( T_1 \rightarrow \text{int} \). Token matches.
  - But + after \( T_1 \) does not match input token *
- Try \( T_1 \rightarrow \text{int} * T_3 \)
  - This will match but + after \( T_1 \) will be unmatched
- Has exhausted the choices for \( T_1 \)
  - Backtrack to choice for \( E_0 \)
Recursive Descent Parsing.

Example (Cont.)

- Try $E_0 \rightarrow T_1$
- Follow same steps as before for $T_1$
- And succeed with $T_1 \rightarrow \text{int} \ast T_2$ and $T_2 \rightarrow \text{int}$
- With the following parse tree

```
        E
       / \  \\
      T   \n     /    |  \\
   int   T  \\
    /     |   \\
  5      \  \\
```


A Recursive Descent Parser.

Preliminaries

- Let TOKEN be the type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES

- Let the global variable `next` point to the next token


A Recursive Descent Parser

(2)

- Define boolean functions that check the token string for a match of
  - A given token terminal
    ```
    bool term(TOKEN tok) { return *next++ == tok; }
    ```
  - A given production of S (the nth)
    ```
    bool Sn() { … }
    ```
  - Any production of S:
    ```
    bool S() { … }
    ```
  - These functions advance `next`
A Recursive Descent Parser

(3)

- For production $E \rightarrow T$
  
  ```
  bool E1() { return T(); } 
  ```

- For production $E \rightarrow T + E$
  
  ```
  bool E2() { return T() && term(PLUS) && E(); } 
  ```

- For all productions of $E$ (with backtracking)
  
  ```
  bool E() {
      TOKEN *save = next;
      return (next = save, E1())
          || (next = save, E2()); 
  } 
  ```

A Recursive Descent Parser

(4)

- Functions for non-terminal $T$
  
  ```
  bool T1() { return term(OPEN) && E() && term(CLOSE); } 
  
  bool T2() { return term(INT) && term(TIMES) && T(); } 
  
  bool T3() { return term(INT); } 
  ```

  ```
  bool T() {
      TOKEN *save = next;
      return (next = save, T1())
          || (next = save, T2())
          || (next = save, T3()); 
  } 
  ```

Recursive Descent Parsing.

Notes.

- To start the parser
  - Initialize next to point to first token
  - Invoke $E()$

- Notice how this simulates our previous example

- Easy to implement by hand
- But does not always work …
When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S$ a
  ```
  bool S1() { return S() && term(a); }
  bool S() { return S1(); }
  ```
  $S()$ will get into an infinite loop

- A left-recursive grammar has a non-terminal $S$
  $S \rightarrow ^{+} S \alpha$ for some $\alpha$

- Recursive descent does not work in such cases

Elimination of Left Recursion

- Consider the left-recursive grammar
  $S \rightarrow S \alpha | \beta$
  $S$ generates all strings starting with a $\beta$ and followed by a number of $\alpha$

- Can rewrite using right-recursion
  $$S \rightarrow \beta S'$$
  $$S' \rightarrow \alpha S' | \varepsilon$$

More Elimination of Left-Recursion

- In general
  $$S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m$$
  All strings derived from $S$ start with one of $\beta_1, \ldots, \beta_m$ and continue with several instances of $\alpha_1, \ldots, \alpha_n$

- Rewrite as
  $$S \rightarrow \beta_1 S' | \ldots | \beta_m S'$$
  $$S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon$$
General Left Recursion

- The grammar
  \[ S \rightarrow A \alpha | \delta \]
  \[ A \rightarrow S \beta \]
  is also left-recursive because
  \[ S \rightarrow^* S \beta \alpha \]

- This left-recursion can also be eliminated -
  * won't go into this

Summary of Recursive Descent

- Simple and general parsing strategy
  * Left-recursion must be eliminated first
  * … but that can be done automatically
  * Unpopular because of backtracking
    * Thought to be too inefficient

- In practice, backtracking is eliminated by restricting the grammar