CMSC 330: Organization of Programming Languages

Context-Free Grammars: Pushdown Automaton

Reminders

- Project 2 Due Oct. 12

Regular expressions and CFGs

<table>
<thead>
<tr>
<th>Description</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular languages</td>
<td>regular expressions</td>
</tr>
<tr>
<td>context-free languages</td>
<td>context-free grammars</td>
</tr>
</tbody>
</table>

- Programming languages are not regular
  - Matching (an arbitrary number of) brackets so that they are balanced
- Usually almost context-free, with some hacks

Equivalence of DFA and regular grammars

Pushdown Automaton (PDA)

- A pushdown automaton (PDA) is an abstract machine similar to the DFA
  - Has a finite set of states
  - Also has a pushdown stack
- Moves of the PDA are as follows:
  - An input symbol is read and the top symbol on the stack is read
  - Based on both inputs, the machine
    - Enters a new state, and
    - Writes zero or more symbols onto the pushdown stack
    - Or pops zero or more symbols from the stack
  - String accepted if the stack is empty AND the string has ended

Power of PDAs

- PDAs are more powerful than DFAs
  - $a^n b^n$, which cannot be recognized by a DFA, can easily be recognized by the PDA
    - Stack all $a$ symbols and, for each $b$, pop an $a$ off the stack.
    - If the end of input is reached at the same time that the stack becomes empty, the string is accepted
Context-free Grammars in Practice

- Regular expressions are used to turn raw text into a string of tokens
  - E.g., “if”, “then”, “identifier”, etc.
  - Whitespace and comments are simply skipped
  - These tokens are the input for the next phase of compilation
  - Standard tools used include lex and flex
- Many others for Java
- CFGs are used to turn tokens into parse trees
  - This process is called parsing
  - Standard tools used include yacc and bison
- Those trees are then analyzed by the compiler, which eventually produces object code

Parsing

- There are many efficient techniques for turning strings into parse trees
  - They all have strange names, like LL(k), SLR(k), LR(k)...
  - Take CMSC 430 for more details
- We will look at one very simple technique: recursive descent parsing
  - This is a “top-down” parsing algorithm because we’re going to begin at the start symbol and try to produce the string

Example

\[ E \rightarrow \text{id} = n \mid \{ L \} \]
\[ L \rightarrow E ; L | \varepsilon \]
- Here \( n \) is an integer and \( \text{id} \) is an identifier
- One input might be
  - \{ x = 3 ; y = 4 ; ; \}
  - This would get turned into a list of tokens
    ( \( x = 3 ; \{ y = 4 ; ; \} \)
  - And we want to turn it into a parse tree

Example (cont’d)

\[ E \rightarrow \text{id} = n \mid \{ L \} \]
\[ L \rightarrow E ; L | \varepsilon \]
\( \{ x = 3 ; \{ y = 4 ; ; \} \} \)

Parsing Algorithm

- Goal: determine if we can produce a string to be parsed from the grammar’s start symbol
- At each step, we’ll keep track of two facts
  - What tree node are we trying to match?
  - What is the next token (lookahead) of the input string?
- There are three cases:
  - If we’re trying to match a terminal and the next token (lookahead) is that token, then succeed, advance the lookahead, and continue
  - If we’re trying to match a nonterminal then pick which production to apply based on the lookahead
  - Otherwise, fail with a parsing error

Example (cont’d)

\[ E \rightarrow \text{id} = n \mid \{ L \} \]
\[ L \rightarrow E ; L | \varepsilon \]
\( \{ x = 3 ; \{ y = 4 ; ; \} \} \)

lookahead
Definition of First($y$)

- First($y$), for any terminal or nonterminal $y$, is the set of initial terminals of all strings that $y$ may expand to.
  - We’ll use this to decide what production to apply.

Definition of First($y$), cont’d

- For a terminal $a$, First($a$) = { $a$ }
- For a nonterminal $N$:
  - If $N \rightarrow \epsilon$, then add $\epsilon$ to First($N$)
  - If $N \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$, then (note the $\alpha_i$ are all the symbols on the right side of one single production):
    - Add First($\alpha_1 \alpha_2 \ldots \alpha_n$) to First($N$), where First($\alpha_1 \alpha_2 \ldots \alpha_n$) is defined as
      - First($\alpha_i$) if $\epsilon \notin$ First($\alpha_i$)
      - Otherwise (First($\alpha_i$) $\neq \epsilon$) First($\alpha_1 \ldots \alpha_n$)
    - If $\epsilon \in$ First($\alpha_i$) for all $1 \leq i \leq k$, then add $\epsilon$ to First($N$)

Examples

<table>
<thead>
<tr>
<th>$E \rightarrow id = n \mid { L }$</th>
<th>$L \rightarrow E \mid L \mid \epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>First(id) = { id }</td>
<td>First(id) = { id }</td>
</tr>
<tr>
<td>First(“=”) = { “=” }</td>
<td>First(“=”) = { “=” }</td>
</tr>
<tr>
<td>First(n) = { n }</td>
<td>First(n) = { n }</td>
</tr>
<tr>
<td>First(“(”) = { “(” }</td>
<td>First(“)” = { “(” }</td>
</tr>
<tr>
<td>First(“)” = { “)” }</td>
<td>First(“)” = { “)” }</td>
</tr>
<tr>
<td>First(“,”) = { “,” }</td>
<td>First(“,”) = { “,” }</td>
</tr>
<tr>
<td>First(E) = { id, “(” }</td>
<td>First(E) = { id, “(”, “)” }</td>
</tr>
<tr>
<td>First(L) = { id, “(”, “)” }</td>
<td>First(L) = { id, “(”, “)” }</td>
</tr>
</tbody>
</table>

Implementing a Recursive Descent Parser

- For each terminal symbol $a$, create a function `parse_a`, which:
  - If the lookahead is $a$ it consumes the lookahead by advancing the lookahead to the next token, and returns
  - Otherwise fails with a parse error if the lookahead is not $a$
- For each nonterminal $N$, create a function `parse_N`
  - This function is called when we’re trying to parse a part of the input which corresponds to (or can be derived from) $N$
    - `parse_S` for the start symbol $S$ begins the process

Example

```ocaml
let parse_term t =
  if !lookahead = t
  then lookahead := <next token>
  else raise <Parse error>

let rec parse_E () =
  if lookahead = 'id' then begin
    parse_term 'id';
    parse_term '=';
    parse_term 'n'
  end
  else if lookahead = '{' then begin
    parse_term '{';
    parse_L ();
    parse_term '}'
  end
  else raise <Parse error>;
```
(not quite valid OCaml)

Example (cont’d)

\[ E \rightarrow \text{id} = n \mid \{ L \} \]
\[ L \rightarrow E \mid L \mid \epsilon \]

\[
\text{and parse}_L () = \begin{cases} 
\text{id} = n \mid \{ \text{parse}_L \} ; \text{parse}_E () ; \text{parse}_\text{term} ; ; \text{parse}_L () \\ (* \text{else return (not an error)} *) 
\end{cases}
\]

Things to Notice

- If you draw the execution trace of the parser as a tree, then you get the parse tree.
- This parsing strategy may fail on certain grammars because the First sets overlap.
  - This doesn’t mean the grammar is not usable in a parser, just not in this type of parser.
- Consider parsing the grammar \( E \rightarrow n + E \mid n \)
  - First(\( E \)) = first(n), so we can’t use this technique.
  - Exercise: Rewrite this grammar so it becomes amenable to our parsing technique.
- This is a predictive parser because we use the lookahead to determine exactly which production to use.

More on Limitations

- How about the grammar \( S \rightarrow Sa \mid \epsilon \)
  - First(\( Sa \)) = a, so we’re ok as far as which production.
  - But the body of parse_\( S() \) has an infinite loop.
  - This technique cannot handle left-recursion.
- Exercise: rewrite this grammar to be right-recursive.

Expr Grammar for Top-Down Parsing

\[ E \rightarrow T E' \]
\[ E' \rightarrow \epsilon \mid + E \]
\[ T \rightarrow P T' \]
\[ T' \rightarrow \epsilon \mid * T \]
\[ P \rightarrow n \mid ( E ) \]

- Notice we can always decide what production to choose with only one symbol of lookahead.

Tradeoffs with Other Approaches

- Recursive descent parsers are easy to write.
  - The formal definition is a little clunky, but if you follow the code then it’s almost what you might have done if you weren’t told about grammars formally.
  - They’re unable to handle certain kinds of grammars.
- More powerful techniques need tool support, such as yacc and bison (which can be slower).
- Recursive descent is good for a quick hack.
  - Though using the tools is pretty fast if you’re familiar with them.

General parsing algorithms

- As with NFA, we can also have a NDPDA.
  - NDPDA are more powerful than DPDA.
  - NDPDA can recognize even length palindromes over \{0,1\}*, but a DPDA cannot. Why? (Hint: Consider palindromes over \{0,1\}[20,1])
  - (Remember that DFA and NFA do accept the same sets.)
- Knuth in 1965 showed that the deterministic PDAs were equivalent to a class of grammars called LR(k) [Left-to-right parsing with k symbol lookahead].
  - Create a PDA that decides whether to stack the next symbol or pop a symbol off the stack by looking k symbols ahead.
  - This is a deterministic process, and for k=1 is efficient.
- LR(k), SLR(k) [Simple LR(k)], and LALR(k) [Lookahead LR(k)] are all techniques used today to build efficient parsers.
  - Recursive descent is a form of LL(k) parsing.
  - More in CMSC 430 …
What’s Wrong with Parse Trees?

- Parse trees contain too much information
  - E.g., they have parentheses and they have extra nonterminals for precedence
  - This extra stuff is needed for parsing
- But when we want to reason about languages, it gets in the way (it’s too much detail)

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts

ASTs (cont’d)

- Intuitively, ASTs correspond to the data structure you’d use to represent strings in the language
  - Note that grammars describe trees (so do OCaml datatypes which we’ll see later)
  - $E \rightarrow a | b | c | E+E | E-E | E*E | (E)$

The Compilation Process

Producing an AST

- To produce an AST, we modify the parse() functions to construct the AST along the way

Producing an AST (cont’d)

```ocaml
let rec parse_E () =
  if lookahead = 'id' then
    let id = parse_term 'id' in
    let _ = parse_term '=' in
    let n = parse_term 'n' in
    Assn(id, int_of_string n)
  else if lookahead = '{' then begin
    let _ = parse_term '{' in
    let l = parse_L () in
    let _ = parse_term '}' in
    Block l
  end
  else raise Parse error;
```
Producing an AST (cont’d)

```plaintext
type ast =
  Assn of string * int
| Block of ast list

and parse_L () =
  if lookahead = 'id' then
    let e = parse_E () in
    let _ = parse_term ';' in
    let l = parse_L () in
    e :: l
  else []
```

COSC 333