CMSC 330: Organization of Programming Languages

Context-Free Grammars:
Pushdown Automaton

Reminders

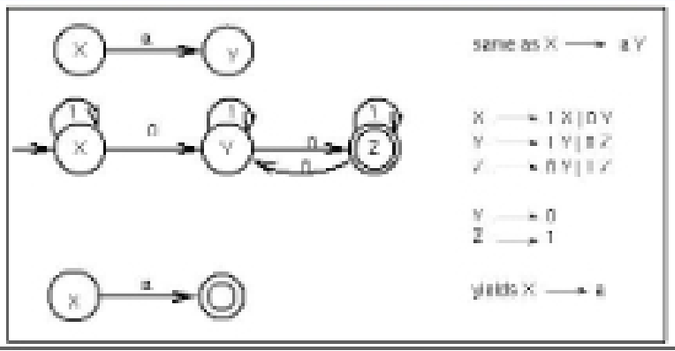
- Project 2 Due Oct. 12

Regular expressions and CFGs

	Description	Machine
regular languages	regular expressions	DFAs, NFAs
context-free languages	context-free grammars	pushdown automata (PDAs)

- 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 → id = n | { L }

L → E ; L | ε

- Here n is an integer and id is an identifier

- One input might be
  - { x = 3; { y = 4; }; }
  - This would get turned into a list of tokens

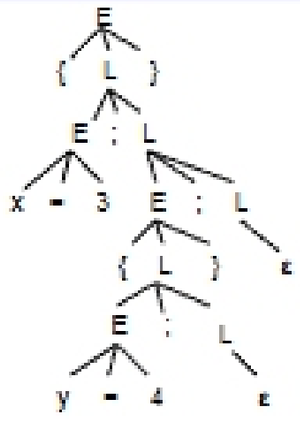
    { x = 3 ; { y = 4 ; } ; }
  - And we want to turn it into a parse tree

Example (cont'd)

E → id = n | { L }

L → E ; L | ε

{ x = 3; { y = 4; }; }



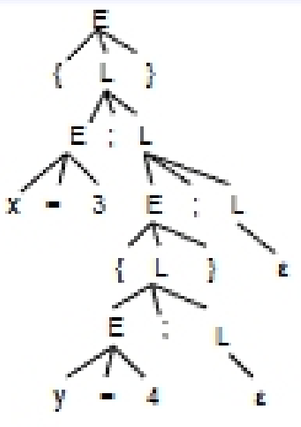
Parsing Algorithm

- 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 → id = n | { L }

L → E ; L | ε



{ x = 3 ; { y = 4 ; } ; }

lookahead

Definition of First(γ)

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

Definition of First(γ), cont'd

- For a terminal a, First(a) = { a }
- For a nonterminal N:
  - If $N \rightarrow \varepsilon$, then add ε to First(N)
  - If $N \rightarrow \alpha_1\ \alpha_2 \ldots \alpha_n$, then (note the α_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(α_1) if $\varepsilon \notin$ First(α_1)
      - Otherwise (First(α_1) – ε)∪ First($\alpha_2 \ldots \alpha_n$)
    - If $\varepsilon \in$ First(α_i) for all i, $1 \le i \le k$, then add ε to First(N)

Examples

$E \rightarrow$ id = n | { L }	$E \rightarrow$ id = n | { L } | ε
$L \rightarrow E$; L | ε	$L \rightarrow E$; L | ε
First(id) = { id }	First(id) = { id }
First("=") = { "=" }	First("=") = { "=" }
First(n) = { n }	First(n) = { n }
First("{")= { "{" }	First("{")= { "{" }
First("}")= { "}" }	First("}")= { "}" }
First(";")= { ";" }	First(";")= { ";" }
First(E) = { id, "{" }	First(E) = { id, "{", ε }
First(L) = { id, "{", ε }	First(L) = { id, "{", ";", ε }

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

Implementing a Recursive Descent Parser, con't.

- The body of parse_N for a nonterminal N does the following:
  - Let $N \rightarrow \beta_1 \mid \ldots \mid \beta_k$ be the productions of N
    - Here β_i is the entire right side of a production- a sequence of terminals and nonterminals
  - Pick the production $N \rightarrow \beta_i$ such that the lookahead is in First(β_i)
    - It must be that First(β_i) ∩ First(β_j) = ∅ for $i \neq j$
    - If there is no such production, but $N \rightarrow \varepsilon$ then return
    - Otherwise, then fail with a parse error
  - Suppose $\beta_i = \alpha_1\ \alpha_2 \ldots \alpha_n$. Then call parse_$\alpha_1$(); ... ; parse_$\alpha_n$() to match the expected right-hand side, and return

Example

$E \rightarrow$ id = n \| { L }

$L \rightarrow E$; L \| ε

```
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)