CMSC 330: Organization of Programming Languages

Context-Free Grammars:
Pushdown Automaton

Reminders

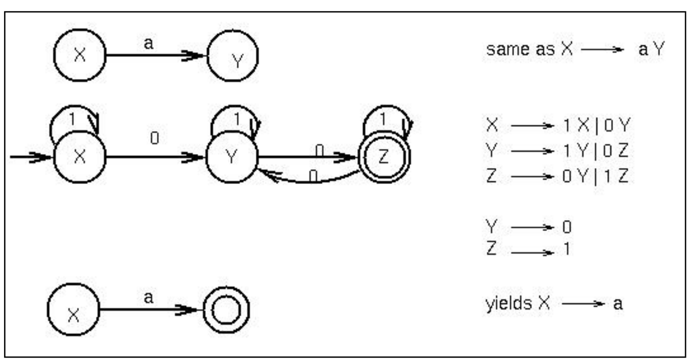
- Project 2 Due Oct. 12

Regular expressions and CFGs

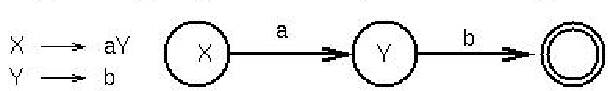
	Description	Machine
regular languages	regular expressions	DFAs, NFAs
context-free languages	context-free grammars	pushdown automata (PDAs)

- 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



To go from regular grammar to FSA, make the following transformations:



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

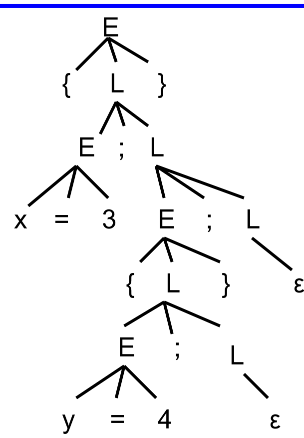
    { x = 3 ; { y = 4 ; } ; }
  - And we want to turn it into a parse tree

Example (cont'd)

E → id = n | { L }

L → E ; L | ε

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

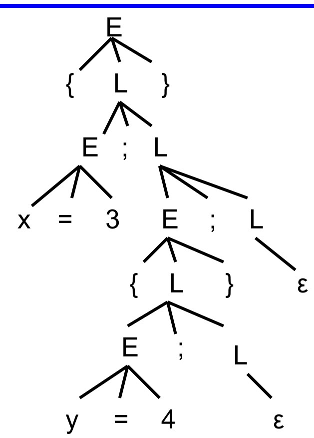


Parsing Algorithm

- 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 $N \to \varepsilon$, then add ε to First(N)
  - If $N \to \alpha_1\ \alpha_2\ \ldots\ \alpha_n$, 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
      - $First(\alpha_1)$ if $\varepsilon \notin First(\alpha_1)$
      - Otherwise $(First(\alpha_1) - \varepsilon) \cup First(\alpha_2\ \ldots\ \alpha_n)$
    - If $\varepsilon \in First(\alpha_i)$ for all i, $1 \leq i \leq k$, then add ε to First(N)

Examples

E → id = n | { L }
L → E ; L | ε

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First("}")= { "}" }
First(";")= { ";" }
First(E) = { id, "{" }
First(L) = { id, "{", ε }

E → id = n | { L } | ε
L → E ; L | ε

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First("}")= { "}" }
First(";")= { ";" }
First(E) = { 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 $First(\beta_i)$
    - It must be that $First(\beta_i) \cap First(\beta_j) = \varnothing$ 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

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

$L \rightarrow E ; L \mid \varepsilon$

```
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 → id = n | { L }

L → E ; L | ε

mutually recursive with previous `let rec`

```
and parse_L () =
  if lookahead = 'id'|| lookahead = '{' then begin
    parse_E ();
    parse_term ';';
    parse_L ()
  end
  (* else return (not an error) *)
```


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 → n + E | n
  - First(E) = n = 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 $S \rightarrow Sa \mid \varepsilon$
  - First(Sa) = a, so we're ok as far as which production
  - But the body of parse_S() has an infinite loop
    - if (lookahead = "a") then parse_S()
  - This technique cannot handle left-recursion
  - Exercise: rewrite this grammar to be right-recursive

Expr Grammar for Top-Down Parsing

$E \rightarrow T\ E'$
$E' \rightarrow \varepsilon \mid +\ E$
$T \rightarrow P\ \ T'$
$T' \rightarrow \varepsilon \mid *\ T$
$P \rightarrow n \ \mid\ (\ E\ )$

- 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 $\{0,1\}^*2\{0,1\}^*$)
  - (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 …

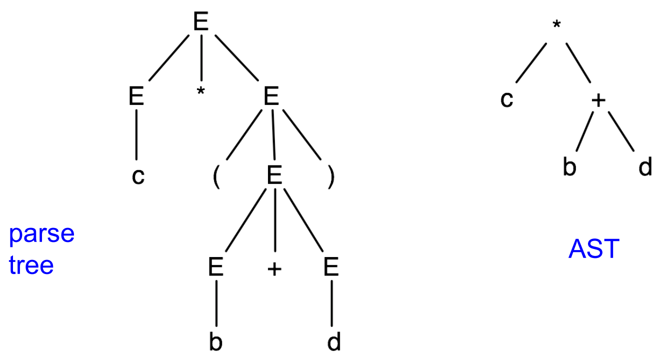
What's Wrong with Parse Trees?

- 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)

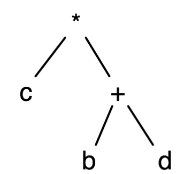
Abstract Syntax Trees (ASTs)

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

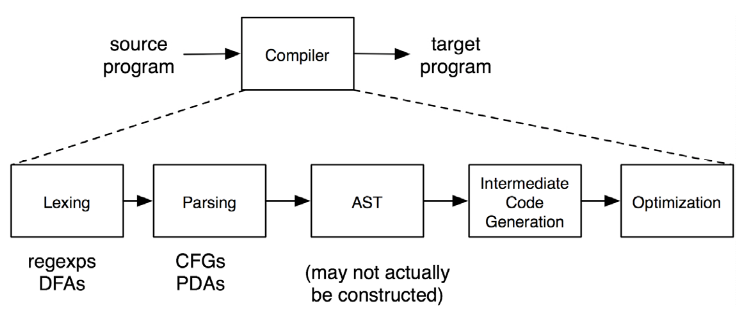


ASTs (cont'd)

- 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 → 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)

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

```
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)

```
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 []
```