Syntax

Amer Diwan

What is syntax?

• Syntax
  – Determines the form of programs
• Semantics
  – Determines the meaning of programs
  – Not all syntactically correct programs are “meaningful”
  – e.g., int i; i.m();
    is syntactically correct but semantically meaningless
Lexical analysis

- **What are tokens?**
  - Variables (x, i, theCount, ...)
  - Keywords (while, begin, ...)
  - Numbers (10, 1.5E10, ...)
  - Other symbols ({, ...)

- **How do languages specify legal tokens?**
  (what is a legal variable name?)
Specifying tokens with regular expressions

• What is a regular expression?
  – A character (e.g., opening_brace = { )
  – An empty string, denoted ?
  – Two regular expressions next to each other (e.g.,
    auto_incr = plus plus)
  – Two regular expressions separated by a vertical bar
    (e.g., binary_op = plus | minus)
  – A regular expression followed by a kleene-star
    (e.g., integer = digit digit*)

Examples of regular expressions

• variable_name =
  (a|b|...|z|A|B|...|Z)(a|b|...|z|A|B|...|Z|0|...|9)*
  What kinds of variable names does this allow?

• digit = 0|1|...|9
  unsigned_integer = digit digit*
  unsigned_number =
  unsigned_integer((. unsigned_integer) | ?)
Recognizing tokens: Ad-hoc approach (small grammars)

- `getWord(InputStream is) {...}
- `getNumber(InputStream is) {...}
- `matchKeyword(String s) {
  if (s.equal("BEGIN")) ...
  else if (s.equal("END")) ...
  ...
}
- `if is_num(next_char)
  return new Number(getNumber())
else { word = getWord();
  if matchKeyword(word) then ...
  else return new Identifier(word)}`

Recognizing tokens (larger grammars)

- Build (or have a tool build) a scanner based on a DFA

  ![Diagram]

  - Want the longest token
  - `myVar` and not `m` or `my` or `myV` or ...
Language choices may complicate lexical analysis

- **FORTRAN (pre ‘90)**
  - `DO 5 I = 1.25` assigns a value 1.25 to variable DO5I (spaces are ignored)
  - May lead to programmer errors!
- **Pascal:** is 4.
  - Start of a floating point number (4.51)?
  - Half way through a subrange (4..20)?
  - Need to look ahead 2 characters

Parsing

- **What are the possible statements of a language?**
  - Addition `(a+b)`
  - While loop `(while (b) { ... })`
  - For loop `(for (i = 0; i < 10; ++i) { ... })`
  - Class declaration `(class S extends T {...})`
  - ...
- **How do languages specify legal statements?**
Context-free grammars

- regular expressions + recursion
- $\text{expr} \rightarrow \text{identifier} | \text{number} | - \text{expr} \\
  | \text{expr} \text{operator} \text{expr}$
- $\text{operator} \rightarrow + | - | * | /$
- Symbols that appear on lhs are non-terminals
- Symbols that appear only on rhs are terminals
  - A scanner returns a stream of terminals to the parser

Parse tree for $-x+y$

Does this match what you would mean with $-x+y$?
Another parse for \(-x+y\)

```
expr -> identifier | number | - expr
    | expr operator expr
operator -> + | - | * | /
```

 identifier(y) 
 expr 
 + 
 expr 

- expr 

Does this match what you would mean with \(-x+y\)?

---

Ambiguity in grammars

- When more than one production can be used
  - Resolution mechanisms:
    - Precedence (* has higher precedence than +)
    - Associativity (10 - 4 - 3 means \((10 - 4) - 3\))
- Let’s rewrite the grammar to eliminate ambiguities
10-4-3

Desired parse:

```
  a nonterminal
     /        \
a nonterminal  \
  number (10) - number (4) - number (3)
```

“-” groups to the “left” (left associative)
instead of `expr -> expr operator expr`, have
`expr -> expr operator term | term`
`term -> identifier | number`

Note how the higher precedence operators come lower in the derivation tree
expr -> expr add_op term | term
term -> term mult_op factor | factor
factor -> identifier | number | - factor | ( expression)
Parsing using the grammar
Top down parsing

• Predict non-terminal and match
  – id_list -> id id_list_tail
    id_list_tail -> , id id_list_tail | ;
    
    match_id_list() {
      match_id(); match_id_list_tail(); }

    match_id_list_tail() {
      if (cur_token == comma) { 
        match_comma(); match_id(); match_id_list_tail(); }
      else if (cur_token == semicolon) {
        match_semicolon();
      }
      else error();
    }

  – More intuitive (when grammar is suitable)

Parsing
What next?

• What does the parser in the previous slide do?
  – x, y, z: Yup!
  – x,, y, z: Nope!
  – x y, z: Nope!

• What else needs to happen in a parser?
  – Build a parse or abstract syntax tree for later use
Building an AST

- E.g., AST for:
  `<ul><li>item 1</li><li>item 2</li></ul>`

Top down parsing
When is the grammar suitable?

- When you can predict the next production with a fixed (small) number of lookaheads
- How many lookaheads do we need?
  - `id_list` -> `id id_list_tail`
  - `id_list_tail` -> `, id id_list_tail` | `;`
  - `id_list` -> `id_list` , `id` | `;`
  - `expr` -> `expr add_op term` | `term`
  - `term` -> `term mult_op factor` | `factor`
  - `factor` -> `identifier` | `number` | `- factor` | `(expression)`
An example

• A -> Aa | Aab | c

Another example

• S -> a S a | epsilon
Bottom-up parsing

- Look at token stream and construct non-terminals
  - id_list -> id id_list_tail
    id_list_tail -> , id id_list_tail | ;
  - Parsing A,B
    - “A”: doesn’t match the rhs of any production
    - “A,”: ditto
    - “A,B”: ditto
    - “A,B;”: “;” matches a rhs of id_list_tail
    - “A,B id_list_tail”: ditto
    - “A id_list_tail”: matches rhs of id_list. Done!

More on bottom-up parsing

- Can handle more grammars than top-down since there is no prediction
- We will look at only top-down in this course
Language design and parsing

• Language design can have a significant effect on the complexity of parsers
  – Languages that are not amenable to straightforward parsing may also interfere with human readability
  – e.g., Pascal if-then-else statements
    if (x) then
      if (y) then
        write(“y’s then”);
      else
        write (“whose else??”);

Relationship to other courses

• CSCI 4555:
  – More emphasis on using parser generation tools for larger grammars than we will use here
Relationship to readings

• Reading covers parsing and lexical analysis in more detail
  – Gives more examples for both
  – Discusses how to resolve ambiguities in grammars

• You will need to know the above for homework and quizzes

Reading for next class

• Sections 3.1, 3.2, 3.3, 3.5, 3.6
  (we will cover Section 3.4 with Chapter 8)