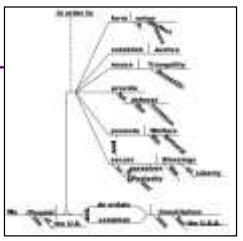
Syntactic Analysis



David Notkin
Autumn Quarter 2008

Syntactic Analysis / Parsing

- Goal: Convert token stream to abstract syntax tree
- Abstract syntax tree (AST)
  - Captures the structural features of the program
  - Primary data structure for remainder of analysis
- Three Part Plan
  - Study how context-free grammars specify syntax
  - Study algorithms for parsing / building ASTs
  - Study the miniJava Implementation

But first, some fun quotations

- At least for the people who send me mail about a new language that they're designing, the general advice is: do it to learn about how to write a compiler. [Dennis Ritchie]
- Thompson and Ritchie were among the first to realize that hardware and compiler technology had become good enough that an entire operating system could be written in C, and by 1978 the whole environment had been successfully ported to several machines of different types. [Eric Raymond]
- If you have a procedure with 10 parameters, you probably missed some. [Perlis]
- It goes against the grain of modern education to teach students to program. What fun is there to making plans, acquiring discipline, organizing thoughts, devoting attention to detail, and learning to be self critical. [Perlis]
- It is easier to change the specification to fit the program than vice versa. [Perlis]
- There are two ways to write error-free programs; only the third one works. [Perlis]

Context-free Grammars (CFGs)

- Compromise between
  - Regular expressions and their lack ofrecursive structure
  - General grammars, unneeded power, undecidable♦
- Context-free grammars
  - \+ Powerful enough to describe nesting, recursion
  - \+ Easy to parse; generally efficient
  - – Cannot capture semantics, as in, “variable must be declared,” requiring later semantic pass
  - – Can be ambiguous♥

Terminology

- **Terminals** – alphabet of language defined by CFG
- **Nonterminals** – symbols defined in terms of terminals and nonterminals
- **Productions** – rules for how a nonterminal (lefthand side, lhs) is defined in terms of a (possibly empty) sequence of terminals and nonterminals
  - Multiple alternative productions allowed for a nonterminal
- **Start symbol** – root of the defining language

```
Program ::= Stmt
Stmt ::= if ( Expr ) then Stmt else Stmt
Stmt ::= while ( Expr ) do Stmt
```


Derivations and Parse Trees

- **Derivation**: a sequence of expansion steps, beginning with a start symbol and leading to a sequence of terminals
- **Parsing**: inverse of derivation
  - Given a sequence of terminals (i.e., tokens) recover the nonterminals representing structure
- Can represent a derivation as a parse tree, that is, the concrete syntax tree

Example Grammar

```
E  ::= E op E | - E | ( E ) | id
op ::= + | - | * | /
```

```
    a    *    (    b    +    -    c    )
```


Ambiguity

- Some grammars are ambiguous: multiple distinct parse trees for the same terminal string
  - The "hi2bob" lexing example was essentially the same problem
- Since the structure of the parse tree captures much of the meaning of the program, ambiguity implies multiple possible meanings for the same program
- This isn't good for programming languages: if the programmer wrote an ambiguous program, the decision of the compiler writer would define the semantics of the program
- "The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do." [Ted Nelson]

Famous Ambiguity: "Dangling Else"

```
Stmt ::= ... |
     if ( Expr ) Stmt |
     if ( Expr ) Stmt else Stmt
```

if (e_1) if (e_2) s_1 else s_2 : if (e_1) if (e_2) s_1 else s_2

Resolving Ambiguity: first two options

- Option 1: a meta-rule such as "else associates with closest previous if"
  - works, keeps original grammar intact
  - ad hoc and informal
- Option 2: rewrite the grammar to avoid ambiguity
  - formal, no additional rules beyond syntax
  - sometimes obscures original grammar

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ... |
                  if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
                  if ( Expr ) MatchedStmt else UnmatchedStmt
```


Resolving Ambiguity Example

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ... |
       if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
       if ( Expr ) MatchedStmt else UnmatchedStmt
```

if (e_1) if (e_2) s_1 else s_2

Resolving Ambiguity: third option

- Redesign the language to remove the ambiguity
  - formal, clear, elegant
  - allows sequence of `Stmts` in **`then`** and **`else`** branches, no braces are needed
  - extra **`end`** required for every **`if`**

```
Stmt ::= ... |
         if Expr then Stmt end |
         if Expr then Stmt else Stmt end
```

Expression example: reprise

```
E  ::= E Op E | - E | ( E ) | id
Op ::= + | - | * | /

a  +  b  *  c  :  a  +  b  *  c
```


Resolving Ambiguity (Option 1)

- Add some meta-rules, e.g. precedence and associativity rules

Example:

```
E ::= E Op E | - E | E ++
      | ( E ) | id
Op::= + | - | * | / | %
      | ** | == | < | &&
      | ||
```

Operator	Preced	Assoc
++ [postfix]	Highest	Left
– [prefix]		Right
**		Right
*, /, %		Left
+, –		Left
==, <		None
&&		Left
||	Lowest	Left

Removing Ambiguity (Option 2)

- Modify the grammar to explicitly resolve the ambiguity
  - create a nonterminal for each precedence level
  - expr is lowest precedence nonterminal
    - each nonterminal can be rewritten with higher precedence operator, highest precedence operator includes atomic expressions
  - at each precedence level, use
    - left recursion for left-associative operators
    - right recursion for right-associative operators
    - no recursion for non-associative operators

Redone Example

```
E  ::= E0
E0 ::= E0 || E1 | E1                left associative
E1 ::= E1 && E2 | E2                left associative
E2 ::= E3 (== | <) E3 | E3          non associative
E3 ::= E3 (+ | -) E4 | E4           left associative
E4 ::= E4 (* | / | %) E5 | E5       left associative
E5 ::= E6 ** E5 | E6                right associative
E6 ::= - E6 | E7                    right associative
E7 ::= E7 ++ | E8                   left associative
E8 ::= id | ( E )
```

Designing A Grammar

- Accurate
- Unambiguous
- Formal
- Readable, Clear
- Parsable by a particular algorithm
  - Top down parser ==> LL(k) Grammar
  - Bottom up Parser ==> LR(k) Grammar
- Design to implementation relatively straightforward
  - By hand
  - By automatic tools

Brainstorm: how to parse?