Context Free Grammars and Syntax

- Making syntactic sense of a stream of tokens
- First we need terminology and theory

Definition of Context Free Grammar

- **Terminals** - An alphabet (like regular expressions, only now the symbols are whole tokens, not characters), including $\varepsilon$
- **Non-terminals** - A set of *names* for structures (like `statement`, `expression`, `definition` – sometimes called the *variables* of the grammar)
- **Production rules** - The grammar *rules* specify substitutions of non-terminals by strings of terminals and non-terminals – these express the structure of the names
- A *start* symbol (the name of the most general structure — `compilation_unit` in C or Java)
CFGs and Languages

- Think of the grammar rules as a series of definitions of the elements of the languages
  - Often have multiple choices for definitions
- Beginning with start symbol (the program), choose a definition (rule)
- Then choose definitions for each new structure element that has appeared
- Repeat this process until only terminals remain

- You have generated a program from the grammar!

Basic Example: Simple integer arithmetic expressions

In what way does such a CFG differ from a regular expression?

- $\text{digit} = 0|1|\ldots|9$
- $\text{number} = \text{digit} \\text{digit}^*$

Recursive rules

“Base” rule

2 non-terminals

6 terminals

6 productions (3 on each line)
Recursion

- Recursion is a natural way to define programs
  - Expressions are made up of expressions
  - Structure of nested blocks is recursively defined…
  - Data structures are collections of data structures…
  - Multi-dimensional arrays are arrays of arrays…
  - Nested parentheses, braces, brackets…

- Thus, we need a CFG

CFGs compared to DFAs

- DFAs use finite memory (the number of states)
- CFGs require unbounded memory (a stack of indeterminate length)
- Real programming languages are too complicated for regular expressions (DFAs), but not too complex for a grammar
CFG as a Program Recognizer

- CFGs can generate programs, but how to determine if a given program is generated?
  - This is the activity of parsing
- Requires backtracking
- Algorithms use a stack – many different approaches
- More difficult to eliminate the non-determinism
- Two basic flavors – top down and bottom up parsing

Parse trees and grammars

Express matching of a string “(34 - 3)*42” by a derivation:

1. \( exp \Rightarrow exp \ op \ exp \) [\( exp \Rightarrow exp \ op \ exp \)]
2. \( \Rightarrow exp \ op \ number \) [\( exp \Rightarrow number \)]
3. \( \Rightarrow exp \ op \ number \) [\( op \Rightarrow * \)]
4. \( \Rightarrow (exp) \ op \ number \) [\( exp \Rightarrow (exp) \)]
5. \( \Rightarrow (exp \ op \ exp) \ op \ number \) [\( exp \Rightarrow exp \ op \ exp \)]
6. \( \Rightarrow (exp \ op \ number) \ op \ number \) [\( exp \Rightarrow number \)]
7. \( \Rightarrow (exp \ op \ number) \ op \ number \) [\( op \Rightarrow * \)]
8. \( \Rightarrow (number \ - \ number) \ op \ number \) [\( exp \Rightarrow number \)]
Abstract the structure of a derivation to a parse tree:

Derivations can vary, even when the parse tree doesn’t:

leftmost derivation (previous - rightmost)

1. exp ⇒ exp op exp  [exp → exp op exp]
2. ⇒ (exp) op exp  [exp → ( exp ]
3. ⇒ (exp op exp) op exp  [exp → exp op exp]
4. ⇒ (number op exp) op exp  [exp → number]
5. ⇒ (number - exp) op exp  [op → ]
6. ⇒ (number - number) op exp  [exp → number]
7. ⇒ (number - number) * exp  [op → *]
8. ⇒ (number - number) * number  [exp → number]
**Parse tree for leftmost derivation:**

```
    exp
   /   \
  2 exp 8 exp
   |     |     
  7 op   4 exp
   |      |     
  6 exp 5 op
       |     |
  number - number
```

Same tree structure as rightmost!
Order of construction different, but end result is the same.

**Derivations and Parsing**

A leftmost derivation corresponds to a (top-down) preorder traversal of the parse tree.

A rightmost derivation corresponds to a (bottom-up) postorder traversal, but in reverse.

Top-down parsers construct leftmost derivations.
(LL = Left-to-right traversal of input, constructing a Leftmost derivation)

Bottom-up parsers construct rightmost derivations in reverse order.
(LR = Left-to-right traversal of input, constructing a Rightmost derivation)
What is the parse tree if there are no parens: \(34 - 3 \times 42\)?

The grammar is **ambiguous**, but is it a problem?  
**Yes ... Semantics!**

---

**Principle of Syntax-directed Semantics**

- Parse tree is the basic model
- Semantic content is attached to the tree
- Thus the tree should reflect the structure of the eventual semantics

- Could describe as *semantics-based syntax*
Sources of Ambiguity

- Associativity and precedence of operators
- Sequencing (e.g., lists)
- Extent of a nested structure (dangling else)
- “Obscure” recursion (unusual)
  - \( exp \rightarrow exp \ exp \)

Dealing with ambiguity

- Change the language
  - Only feasible if language is being designed
  - Only makes sense if language is improved
- Change the grammar (but not the language!)
- Disambiguating rules
  - Almost like extensions of the grammar
- Can all ambiguity be removed?
  - Backtracking can handle it, but expense is great
  - Some specs of language left undefined
Standard Arithmetic Example

exp → exp addop term | term
addop → + | −
term → term mulop factor | factor
mulop → * | /
factor → ( exp ) | number

- This is a precedence “cascade”
- Also handles associativity of 1 + 2 + 3

Repetition and Recursion

- Left recursion: A → A x | y

- Right recursion: A → x A | y
Repetition and Recursion

- Sometimes we care which way recursion goes: operator associativity
- Sometimes we don’t: statement and expression sequences
  - Tree can descend to right or left, but order remains the same
- Parsing always has to pick a way!
- The tree may remove this information

Sequence Examples

- **one** or more stmts *separated* by a semicolon
  \[ stmt-seq \rightarrow stmt \mid stmt-seq \mid stmt \]
- **zero** or more stmts *terminated* by a semicolon
  \[ stmt-seq \rightarrow stmt \mid stmt-seq \mid \varepsilon \]
- **one** or more stmts *separated* by a semicolon
  \[ stmt-seq \rightarrow stmt-seq \mid stmt \mid stmt \]
- **zero** or more stmts *preceded* by a semicolon
  \[ stmt-seq \rightarrow stmt-seq \mid stmt \mid \varepsilon \]
Extended BNF

- Notation for expressing repetition
  - \( stmt-seq \rightarrow stmt \{ ; stmt \} \)
  - Same as left recursive form (repeats on right)
- Obscures structure of parse tree, but appropriate for cases where we don’t care
  - “Flat” lists without associativity
- EBNF also has optional construct
  - \( stmt-seq \rightarrow stmt [ ; stmt-seq ] \)
  - Could use for if’s optional else to avoid ambiguity

Abstract Syntax Trees

- Retain only the essential structure of the parse tree
- Omit parens, cascades, and “don’t-care” repetitive associativity
- Corresponds to actual internal tree structure produced by parser
- Use sibling lists for “don’t care” repetition
  - I.e., don’t retain grouping information
First Example  \((34 - 3) * 42\)

Last Example  \(34 - 3 * 42\)
Another Ambiguity Example

Incorrect attempt to add unary minus:

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp addop term} \mid \text{term} \mid - \text{exp} \\
\text{addop} & \rightarrow + \mid - \\
\text{term} & \rightarrow \text{term mulop factor} \mid \text{factor} \\
\text{mulop} & \rightarrow * \\
\text{factor} & \rightarrow (\text{exp}) \mid \text{number}
\end{align*}
\]

Fixing the grammar

- Better: (but only one at beg. of an exp)
  \[
  \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \mid - \text{term}
  \]
- Or maybe: (many at beginning of term)
  \[
  \begin{align*}
  \text{term} & \rightarrow - \text{term} \mid \text{term1} \\
  \text{term1} & \rightarrow \text{term1 mulop factor} \mid \text{factor}
  \end{align*}
  \]
- Or maybe: (many anywhere)
  \[
  \begin{align*}
  \text{factor} & \rightarrow (\text{exp}) \mid \text{number} \mid - \text{factor}
  \end{align*}
  \]
Another Ambiguity Example

Fragment of a grammar for conditional statements in C (parentheses omitted)

\[
\begin{align*}
\text{if-stmt} & \rightarrow \text{if expr stmt} \\
& \quad \mid \text{if expr stmt else stmt} \\
\text{stmt} & \rightarrow \text{if-stmt} \mid S1 \mid S2
\end{align*}
\]

Consider the statement

\[
\text{if expr if expr S1 else S2}
\]

Dangling else Example

\[
\begin{align*}
\text{if expr if expr S1 else S2} & \quad \text{if expr if expr S1 else S2} \\
\text{if expr if expr S1 else S2} & \quad \text{if expr if expr S1 else S2}
\end{align*}
\]
How to fix dangling else?

- Add the keyword ‘endif’ to constrain the clause
  - But this would change the language and may not be acceptable
- Don’t allow an if without an else
  - This really changes the language
  - ML does this, but if-else is an expression, not a statement there
- Use add hoc rules in the parsing and document the behavior (e.g., else goes with nearest if)

- Fix the grammar (hard, but elegant) …

Unambiguous if-else

\[
\text{matched} \rightarrow \text{if expr matched else matched} \mid S_1 \mid S_2 \\
\text{unmatched} \rightarrow \text{if expr stmt} \\
\phantom{\text{unmatched} \rightarrow \text{if expr stmt}} \mid \text{if expr matched else unmatched} \\
\text{stmt} \rightarrow \text{matched} \mid \text{unmatched}
\]
Another Ambiguity in C

\[
\begin{align*}
\text{cast_expression} & \rightarrow \text{unary_expression} \\
& \quad \mid (\text{type_name}) \text{cast_expression} \\
\text{unary_expression} & \rightarrow \text{postfix_expression} \mid \ldots \\
\text{postfix_expression} & \rightarrow \text{primary_expression} \mid \ldots \\
\text{primary_expression} & \rightarrow \text{IDENTIFIER} \mid \text{CONSTANT} \\
& \quad \mid \text{STRING_LITERAL} \mid \left(\text{expression}\right) \\
\text{type_name} & \rightarrow \ldots \mid \text{TYPE_NAME}
\end{align*}
\]

Example:

\[
\begin{array}{|l|}
\hline
\text{typedef double x;} & \text{int x = 1;} \\
\text{printf("%d\n", (int)(x)-2);} & \text{printf("%d\n", (int)(x)-2);} \\
\hline
\end{array}
\]

Removing the cast ambiguity

- TYPE_IDs must be distinguished from other IDs in the scanner.
- Parser must build the symbol table (at least partially) to indicate whether an ID is a typedef or not.
- Scanner must consult the symbol table; if an ID is found as a typedef, return TYPE_ID, if not return ID.
Object Oriented Hierarchy

- Represent AST with hierarchy of classes
- Abstract base class of ASTnode
  - Derives from Token
- Subclasses for Program, Statements, Expressions
  - Hierarchy of Statements
  - Hierarchy of Expressions

Expression Hierarchy

- ASTnode
  - Exp
    - Literal
    - Variable
    - Binary
      - Arithmetic
      - Relational
Statement Hierarchy

TINY Grammar

```
program → stmt_seq
stmt_seq → stmt_seq ; stmt | stmt
stmt → if exp then stmt_seq end
  | if exp then stmt_seq else stmt_seq end
  | repeat stmt_seq until exp
  | variable := exp
  | read variable
  | write exp
exp → simple_exp < simple_exp | simple_exp = simple_exp | simple_exp
simple_exp → simple_exp * term | simple_exp - term | term
term → term * factor | term / factor | factor
factor → ( exp ) | NUM | variable
variable → ID
```