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

- `digit = 0|1|...|9`
- `number = digit digit*`

Recursion!
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) \( \text{exp} \Rightarrow \text{exp op exp} \) [\( \text{exp} \rightarrow \text{exp op exp} \)]
(2) \( \Rightarrow \text{exp op number} \) [\( \text{exp} \rightarrow \text{number} \)]
(3) \( \Rightarrow \text{exp} \cdot \text{number} \) [\( \text{op} \rightarrow \cdot \)]
(4) \( \Rightarrow (\text{exp}) \cdot \text{number} \) [\( \text{exp} \rightarrow (\text{exp}) \)]
(5) \( \Rightarrow (\text{exp op exp}) \cdot \text{number} \) [\( \text{exp} \rightarrow \text{exp op exp} \)]
(6) \( \Rightarrow (\text{exp op number}) \cdot \text{number} \) [\( \text{exp} \rightarrow \text{number} \)]
(7) \( \Rightarrow (\text{exp - number}) \cdot \text{number} \) [\( \text{op} \rightarrow \ - \)]
(8) \( \Rightarrow (\text{number - number}) \cdot \text{number} \) [\( \text{exp} \rightarrow \text{number} \)]
Abstract the structure of a derivation to a parse tree:

```
  1 exp
   4 exp 3 op 2 exp
      ( 5 exp ) * number
   8 exp 7 op 6 exp
     number - number
```

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

leftmost derivation (previous - rightmost)

(1) exp ⇒ exp op exp [exp \rightarrow exp op exp]
(2) ⇒ (exp) op exp [exp \rightarrow ( exp )]
(3) ⇒ (exp op exp) op exp [exp \rightarrow exp op exp]
(4) ⇒ (number op exp) op exp [exp \rightarrow number]
(5) ⇒ (number - exp) op exp [op \rightarrow -]
(6) ⇒ (number - number) op exp [exp \rightarrow number]
(7) ⇒ (number - number) * exp [op \rightarrow *]
(8) ⇒ (number - number) * number [exp \rightarrow number]
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 * 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

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

- This is a precedence “cascade”
- Also handles associativity of \[ 1 + 2 + 3 \]
Repetition and Recursion

- Left recursion: $A \rightarrow A \ x \ | \ y$

```
  A
 / \
 A x
 /  \
 y
```

- Right recursion: $A \rightarrow x \ A \ | \ y$

```
  A
 / \
 x A
 /  \
 x A
   /  \
   x y
```

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

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  \((34 - 3) \times 42\)

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

Incorrect attempt to add unary minus:

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

Fixing the grammar

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

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

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

Consider the statement

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

Dangling else Example

\[
\begin{align*}
\text{if expr if expr S1 else S2} & \quad | \quad \text{if expr if expr S1 else S2}
\end{align*}
\]
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

\[
\text{matched} \rightarrow \text{if expr matched else matched} \mid S1 \mid S2 \\
\text{unmatched} \rightarrow \text{if expr stmt} \\
\quad \mid \text{if expr matched else unmatched} \\
\text{stmt} \rightarrow \text{matched} \mid \text{unmatched}
\]
Another Ambiguity in C

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

Example:

```c
typedef double x;
printf("%d\n", (int)(x)-2);
int x = 1;
printf("%d\n", (int)(x)-2);
```

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

```
ASTnode
|
Program                  Statement
|
Sequence                Write
|
Read                   If
|
Assign                 Repeat
```

### 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
```