SYNTAX ANALYSIS

Specify legal program formats using **context-free grammar (CFG)**

- Use **Backus-Naur Form (BNF)** as notation.
- Gives precise, readable specification.
- Many CFG’s have efficient **parsers**.
- Parser recognizes syntactically legal programs and rejects illegal ones.

Successful parse also captures **hierarchical** structure of programs (expressions, blocks, etc.).

- Convenient representation for further semantic checking (e.g., types) and for code generation.

We can use another program generator (e.g., yacc or CUP) to generate parser automatically from grammar.

GRAMMARS

A **Context-free Grammar** is described by a set of **productions** (also called **rewrite rules**), e.g.:

- \( stmt \rightarrow \text{if} \ expr \ \text{then} \ stmt \ \text{else} \ stmt \)
- \( expr \rightarrow expr + expr \mid expr * expr \mid (expr) \mid -expr \mid id \)

Grammars contain **terminals** (≡ **tokens**) (e.g. if,+,id) and **non-terminals** (e.g., expr,stmt).

Grammars have a distinguished **start symbol** (ordinarily listed first, e.g., stmt).

The language generated by a grammar is the set of **sentences** (strings) of terminals that can be derived by repeated application of productions, beginning with **start symbol**.

- We write \( L(G) \) for the language generated by grammar \( G \).
A **parse tree** is the graphical representation of a derivation.

Example tree for derivation of sentence $-(\text{id}_X + \text{id}_Y)$:

```
  expr
 / \       
  -  expr
   /  
  ( expr
 /  
 expr + expr
  \    
  \  
 id_X id_Y
```

Each application of a production corresponds to an **internal** node, labeled with a **non-terminal**.

Leaves are labeled with **terminals**, possibly with attributes.

The derived sentence is found by reading leaves left-to-right.

---

**DERIVATIONS**

Can “linearize” a parse tree into a sequence of one-step derivations.

Example:

- $expr \Rightarrow -expr$
- $-(expr) \Rightarrow -(expr + expr)$
- $-(expr + id_Y) \Rightarrow -(id_X + expr)$
- $-(id_X + id_Y)$

or

- $expr \Rightarrow -(id_X + id_Y)$

Here $\Rightarrow$ is pronounced “derives” and $\Rightarrow*$ is pronounced “derives in zero or more steps.”

This example gives a **leftmost derivation**, i.e., at each step, the leftmost non-terminal is replaced.

---

**AMBIGUITY**

**BUT** a given **sentence** in $L(G)$ can have more than one parse tree. Grammars $G$ for which this is true are called **ambiguous**.

Example: with our grammar, the sentence $\text{id}_X + \text{id}_Y * \text{id}_Z$

has two parse trees:

```
  expr
 /  
  + expr
   /  
  expr * expr
    \    
    \    
  id_X id_Y id_Z
```

We might think that the left tree is the “correct” one, but nothing in the grammar says this.
Resolving Ambiguities

Ambiguous grammars can be a significant problem in practice, because we rely on the parse tree to capture the basic structure of a program.

To avoid the problems of ambiguity, we can try to:

- Rewrite grammar
- Use “disambiguating rules” when we implement parser for grammar.

When using these techniques, we must be careful not to change the language generated by our grammar. And often, we must do extra work to make sure that the (unique) parse trees we end up with are the ones we wanted.

A Classic Ambiguity: The “Dangling Else”

Suppose we want else clauses to be optional in if statements. Here’s a possible grammar:

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
\text{ } \quad | \quad \text{if expr then stmt else stmt} \\
\text{ } \quad | \quad \ldots
\]

Given this grammar, a statement of the form

\[
\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2
\]

has two possible parse trees...

Resolving Ambiguity by Rewriting Grammar

For most languages, we want the first tree (else goes with most recent then), but grammar is ambiguous.

Solution: rewrite grammar using new non-terminals \textit{mst} (“matched statement”) and \textit{ust} (“unmatched statement”).

\[
\text{stmt} \rightarrow \text{mst | ust} \\
\text{mst} \rightarrow \text{if expr then mst else mst} \\
\text{ } \quad | \quad \ldots \\
\text{ust} \rightarrow \text{if expr then stmt} \\
\text{ } \quad | \quad \text{if expr then mst else ust}
\]

Now only one parse is possible.

Assuming \(S_1, S_2\) are not unmatched if statements, we get...
Ambiguity in Arithmetic Expressions

A grammar such as

\[ E \rightarrow E + E \mid E - E \mid E \times E \mid E / E \]
\[ E \uparrow E \mid (E) \mid - E \mid id \]

is ambiguous about order of operations.

Want to define

- **Precedence** - which operation is done first (“binds more tightly”)?
- **Associativity** - is

\[ X \ op_1 Y \ op_2 Z \]

equivalent to

\[ (X \ op_1 Y) \ op_2 Z \] (left-associativity)

or to

\[ X \ op_1 (Y \ op_2 Z) \] (right-associativity)

assuming \( op_1 \) and \( op_2 \) have same precedence?

Standard Precedence and Associativity

The “usual” rules (based on common usage in written math) give the following precedences, highest first:

- (unary minus)
- \( \uparrow \) (exponentiation)
- \( * / + - \)

All the binary operators are left-associative except exponentiation (\( \uparrow \)).

Note that these rules are just a matter of **convention**; a programming language designer might choose different ones.

We can handle precedence/associativity information as “side-conditions” to ambiguous grammar when building a parser (by hand or via a parser generator).

Rewriting Arithmetic Grammars

Can build precedence/associativity into grammar using extra non-terminals, each corresponding to a separate level of precedence:

- **atom** → (expr) | id
- **primary** → ~primary | atom
- **factor** → primary \( \uparrow \) factor | primary
- **term** → term * factor | term / factor | factor
- **expr** → expr + term | expr - term | term
Example: $\text{id}_x \ast \text{id}_y + \text{id}_z \uparrow \text{id}_u \uparrow \text{id}_v$

### Extended BNF

Various semi-standard extensions to BNF are often used in language manuals. They allow grammar specifications to avoid explicit recursion and $\epsilon$-productions, by adding **optional** symbols, **repetition**, and **grouping**.

- $[a]$ means $a \mid \epsilon$
- $\{a\}$ means $\epsilon \mid a \mid aa \mid \ldots$
- $(a \mid b)c$ means $ac \mid bc$

To avoid confusion between new meta-symbols and terminals, we often enclose the latter in single quotes:

Example:

- `atom` → `( expr )` | id
- `primary` → `- primary` | atom
- `factor` → `primary` { ↑ primary }
- `term` → { factor ( * | / ) } factor
- `expr` → { term ( + | - ) } term

### Properties of CFG's

Any **regular** language can be described by a CFG.

Example: $(a|b)\ast abb$

- $A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
- $A_1 \rightarrow bA_2$
- $A_2 \rightarrow bA_3$
- $A_3 \rightarrow \epsilon$

**But** we don’t use CFG’s for lexical analysis, because it’s overkill.

Regular expressions are:

- easier to understand
- shorter
- always lead to efficient analyzers

Any **CFL** can be parsed by a computer program, but only some CFL’s can be parsed **efficiently**.

We’ll study both “bottom-up” and “top-down” parsing methods.
Not every language is a CFL.

Example: \( L = \{ wcw \mid w \in (a \mid b)^* \} \) is not CF.

- \( L \) abstracts idea of variable declaration before use.
- So “semantic” analysis (type-checking) uses additional (mostly ad-hoc) techniques.