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.:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if expr then stmt else stmt} \\
\text{expr} & \rightarrow \text{expr} + \text{expr} \mid \text{expr} * \text{expr} \\
& \quad \mid (\text{expr}) \mid -\text{expr} \mid \text{id}
\end{align*}
\]

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 $-(idx + idy)$:

```
expr
  /     \  \
-       expr
    \     /    \
  (      expr
       /     /
expr + expr
```

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.

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

```
expr  ⇒  - expr
      ⇒  -(expr)
      ⇒  -(expr + expr)
      ⇒  -(expr + idy)
      ⇒  -(idx + idy)
```

or

```
expr  ⇒  -(idx + idy)
```

Here $⇒$ is pronounced “derives” and $∗⇒$ 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.

We can define rightmost derivation analogously:

```
expr  ⇒  - expr
      ⇒  -(expr)
      ⇒  -(expr + expr)
      ⇒  -(expr + idy)
      ⇒  -(idx + idy)
```

Every parse tree has a unique leftmost derivation and a unique rightmost derivation.

(These are usually, but not necessarily, different.)

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 $idx + idy * idz$ has two parse trees:

```
expr
  /     \  \
expr + expr
    \     /    \
  idx   expr
       /     /
expr * expr
```

We may think of the left tree as being 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:

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

A CLASSIC AMBIGUITY: THE “DANGLING ELSE”

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

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
\phantom{\text{stmt} \rightarrow} \text{| if expr then stmt else stmt} | ...
\]

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

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

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

\[
\text{stmt} \rightarrow \text{mst | ust} \\
\text{mst} \rightarrow \text{if expr then mst else mst} \\
\phantom{\text{mst} \rightarrow} \text{| ...} \\
\text{ust} \rightarrow \text{if expr then stmt} \\
\phantom{\text{ust} \rightarrow} \text{| if expr then mst else ust}
\]

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

\[ E \to E + E \mid E - E \mid E \ast E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid id \]

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:

\[ atom \to (expr) \mid id \]
\[ primary \to -primary \mid atom \]
\[ factor \to primary \uparrow factor \mid primary \]
\[ term \to term \ast factor \mid term / factor \mid factor \]
\[ expr \to expr + term \mid expr - term \mid term \]
**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.

\[
\begin{align*}
[a] & \quad \text{means} \quad a \mid \epsilon \\
\{a\} & \quad \text{means} \quad \epsilon \mid a \mid aa \mid \ldots \\
(a \mid b)c & \quad \text{means} \quad ac \mid bc
\end{align*}
\]

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

Example:

\[
\begin{align*}
\text{atom} & \quad \rightarrow \quad \mathord{'}(\mathord{'}\; \text{expr}\; \mathord{'}\mathord{')}\mathord{'} \mid \mathord{id} \\
\text{primary} & \quad \rightarrow \quad \mathord{-}\mathord{primary} \mid \mathord{atom} \\
\text{factor} & \quad \rightarrow \quad \mathord{primary} \{ \uparrow \mathord{primary} \} \\
\text{term} & \quad \rightarrow \quad \{ \mathord{factor} (\mathord{\ast} \mid /) \} \mathord{factor} \\
\text{expr} & \quad \rightarrow \quad \{ \mathord{term} (\mathord{+} \mid \mathord{-}) \} \mathord{term}
\end{align*}
\]

**Issues with EBNF**

Must be careful not to change the generated language accidentally when going between EBNF and BNF.

Also, note that EBNF grammar given in example is ambiguous because we’ve lost the associativity information present in the BNF version (even though both generate same language).

Often, a language’s grammar will be given in ambiguous EBNF together with separate informal specifications that resolve the ambiguities.

**Properties of CFG’s**

Any regular language can be described by a CFG.

Example: \((a\mid b)\ast abb\)

\[
\begin{align*}
A_0 & \quad \rightarrow \quad aA_0 \mid bA_0 \mid aA_1 \\
A_1 & \quad \rightarrow \quad bA_2 \\
A_2 & \quad \rightarrow \quad bA_3 \\
A_3 & \quad \rightarrow \quad \epsilon
\end{align*}
\]

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.
CFG's can’t do everything

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.