Syntax Analysis/Parsing

Purpose: stream of tokens ⇒ **abstract syntax tree** (AST)

AST:
- captures hierarchical structure of input program
- primary representation of program, for rest of compiler

Context-free grammars (CFG's)

Syntax specified using CFG's
- RE's not powerful enough
- general grammars (GG's) too powerful

CFG's: convenient compromise
- capture important structural & nesting characteristics
- some properties checked later during semantic analysis

Notation for CFG's:
Extended Backus Normal (Naur) Form (EBNF)

CFG terminology

Terminals: alphabet of language defined by CFG

Nonterminals: symbols defined in terms of terminals and nonterminals

Production: rule for how a nonterminal (l.h.s.) is defined in terms of a finite, possibly empty sequence of terminals & nonterminals
- recursive productions allowed!

Can have multiple productions for same nonterminal
- **alternatives**

Start symbol: root symbol defining language

```
Program ::= Stmt
Stmt    ::= if Expr then Stmt else Stmt end
Stmt    ::= while Expr do Stmt end
```


EBNF description of PL/0 syntax

```
Program      ::=  module Id ; Block Id .
Block        ::=  DeclList begin StmtList end
DeclList     ::=  { Decl ; }
Decl         ::=  ConstDecl | ProcDecl | VarDecl
ConstDecl    ::=  const ConstDeclItem { , ConstDeclItem }
ConstDeclItem::=Id : Type = ConstExpr
ConstExpr    ::=  Id | Integer
VarDecl      ::=  var VarDeclItem { , VarDeclItem }
VarDeclItem::=    Id : Type
ProcDecl     ::=  procedure Id ( [ FormalDecl {, FormalDecl }
                  ] ) ; Block Id
FormalDecl   ::=  Id : Type
Type         ::=  int
StmtList     ::=  { Stmt ; }
Stmt         ::=  CallStmt | AssignStmt | OutStmt | IfStmt |
                  WhileStmt
CallStmt     ::=  Id ( [ Exprs ] )
AssignStmt   ::=  LValue := Expr
LValue       ::=  Id
OutStmt      ::=  output := Expr
IfStmt       ::=  if Test then StmtList end
WhileStmt    ::=  while Test do StmtList end
Test         ::=  odd Sum | Sum Relop Sum
Relop        ::=  <= | <> | < | >= | > | =
Exprs        ::=  Expr { , Expr }
Expr         ::=  Sum
Sum          ::=  Term { (+ | -) Term }
Term         ::=  Factor { (* | /) Factor }
Factor       ::=  - Factor | LValue | Integer | input | ( Expr )
```

Transition diagrams

"Railroad diagrams"

- another, more graphical notation for CFG's
- look like FSA's, where arcs can be labelled with nonterminals as well as terminals

Derivations and parse trees

Derivation: sequence of expansion steps,
beginning with start symbol,
leading to a string of terminals

Parsing: inverse of derivation

- given target string of terminals (a.k.a. tokens), want to recover nonterminals representing structure

Can represent derivation as a **parse tree**

- concrete syntax tree
- abstract syntax tree (AST)

Example grammar

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


Ambiguity

Some grammars are **ambiguous**:

- multiple different parse trees with same final string

Structure of parse tree captures much of meaning of program;
ambiguity $\Rightarrow$ multiple possible meanings for same program

Famous ambiguities: "dangling else"

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

"**if** e_1 **then if** e_2 **then** s_1 **else** s_2"

Resolving the ambiguity

Option 1: add a meta-rule
e.g. "`else` associates with closest previous `if`"

- works, keeps original grammar intact
- ad hoc and informal

Option 2: rewrite the grammar to resolve ambiguity explicitly

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ... |
                  if Expr then MatchedStmt
                          else MatchedStmt
UnmatchedStmt ::= if Expr then Stmt |
                  if Expr then MatchedStmt
                          else UnmatchedStmt
```

- formal, no additional rules beyond syntax
- sometimes obscures original grammar

Option 3: redesign the language to remove the ambiguity

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

- formal, clear, elegant
- allows `StmtList` in `then` and `else` branch, no `begin`/`end` needed
- extra `end` required for every `if`

Another famous ambiguity: expressions

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

"a **+** b ***** c"

Resolving the ambiguity

Option 1: add some meta-rules,
e.g. precedence and associativity rules

Example:

```
E  ::= E Op E | - E | E ! | ( E ) | id
Op ::= + | - | * | / | ^ | = | < | and | or
```

operator	precedence	associativity
postfix !	highest	left
prefix −		right
^		right
*, /		left
+, −		left
=, <		none
and		left
or	lowest	left

Option 2: modify the grammar to explicitly resolve the ambiguity

Strategy:

- 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 exprs
- at each precedence level, use:
  - left recursion for left-associative operators
  - right recursion for right-associative operators
  - no recursion for non-associative operators

Designing a grammar

Concerns:

- accuracy
- readability, clarity
- unambiguity
- limitations of CFG's
- ability to be parsed by particular parsing algorithm
  - top-down parser ⇒ LL(k) grammar
  - bottom-up parser ⇒ LR(k) grammar