Syntactic 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

Plan:
- study how grammars can specify syntax
- study algorithms for constructing ASTs from token streams
- study MiniJava implementation

Context-free grammars (CFG's)

Syntax specified using CFG's
- RE's not powerful enough
  - can't handle nested, recursive structure
- general grammars (GG's) too powerful
  - not decidable ⇒ parser might run forever!

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

Common notation for CFG's:
Extended Backus-Naur Form (EBNF)

Context-free grammar 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

Example, in pure BNF:

```
Program ::= Stmt
Stmt    ::= if ( Expr ) Stmt else Stmt
Stmt    ::= while ( Expr ) Stmt
```

Notational conveniences ⇒ EBNF

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

EBNF description of initial MiniJava syntax

```
Program       ::= MainClassDecl {ClassDecl}
MainClassDecl::= class ID {
                 public static void main
                 ( String [ ] ID ) { {Stmt} } }
ClassDecl     ::= class ID [extends ID] {
                 {ClassVarDecl} {MethodDecl} }
ClassVarDecl ::= Type ID ;
MethodDecl    ::= public Type ID
                 ( [Formal {, Formal}] )
                 { {Stmt} return Expr ; }
Formal        ::= Type ID
Type          ::= int | boolean | ID
Stmt          ::= Type ID = Expr ;
                | { {Stmt} }
                | if ( Expr ) Stmt else Stmt
                | while ( Expr ) Stmt
                | System.out.println ( Expr ) ;
                | ID = Expr ;
Expr          ::= Expr BinOp Expr
                | UnOp Expr
                | Expr . ID ( [Expr {, Expr}] )
                | new ID ( )
                | ID | this
                | Integer | null | true | false
                | ( Expr )
BinOp         ::= + | - | * | /
                | < | <= | >= | > | == | != | &&
UnOp          ::= - | !
```


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

Example grammar

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


Ambiguity

Some grammars are **ambiguous**:

- multiple distinct parse trees with same final string

Structure of parse tree captures much of meaning of program;
ambiguity ⇒ multiple possible meanings for same program

Famous ambiguities: "dangling else"

```
Stmt ::= ...
       | if ( Expr ) Stmt
       | if ( Expr ) Stmt else Stmt
```

"**if (**e_1**) if (**e_2**)** 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

Resolving the ambiguity (cont.)

Option 2: rewrite the grammar to resolve ambiguity explicitly

```
Stmt          ::= MatchedStmt | UnmatchedStmt
MatchedStmt   ::= ...
                | if ( Expr ) MatchedStmt
                           else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt
                | if ( Expr ) MatchedStmt
                           else UnmatchedStmt
```

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

Resolving the ambiguity (cont.)

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 sequence of `Stmt`s in `then` and `else` branches, no `{`, `}` 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 ::= + | - | * | / | % | ** | == | < | && | ||
```

operator	precedence	associativity
postfix ++	highest	left
prefix –		right
** (expon.)		right
*, /, %		left
+, –		left
==, <		none
&&		left
||	lowest	left

Resolving the ambiguity (cont.)

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

Example, redone

```
E  ::= E0
E0 ::= E0 || E1 | E1                 left associative
E1 ::= E1 && E2 | E2                 left associative
E2 ::= E3 (== | <) E3                non associative
E3 ::= E3 (+ | -) E4 | E4            left associative
E4 ::= E4 (* | / | %) E5 | E5        left associative
E5 ::= E6 ** E5 | E6                 right associative
E6 ::= - E6 | E7                     right associative
E7 ::= E7 ++ | E8                    left associative
E8 ::= id | ( E )
```

Resolving the ambiguity (cont.)

Option 3: redesign the language to remove the ambiguity

E.g. Lisp/Scheme syntax, which uses **prefix** form consistently for both functions and operators

- no precedence or associativity rules needed

```
E  ::= ( E {E} ) | Op | id | int
Op ::= + | - | * | / | % | ** | == | < | && | ||
```

`(* (+ a b) c)` vs. `(+ a (* b c))`

Designing a grammar

Concerns:

- accuracy
- unambiguity
- formality
- readability, clarity
- ability to be parsed by particular parsing algorithm
  - top-down parser ⇒ LL(k) grammar
  - bottom-up parser ⇒ LR(k) grammar
- ability to be implemented using a particular strategy
  - by hand
  - by automatic tools

Parsing algorithms

Given grammar, want to parse input programs

- check legality
- produce AST representing structure
- be efficient

Kinds of parsing algorithms:

- top-down
- bottom-up

Top-down parsing

Build parse tree for input program from the top (start symbol) down to leaves (terminals)

Basic issue:

- when "expanding" a nonterminal with some r.h.s., how to pick which r.h.s.?

E.g.

```
Stmt    ::= Assign | Call | If | While
Assign  ::= Id = Expr ;
Call    ::= Id ( Expr {, Expr} ) ;
If      ::= if ( Test ) Stmt
          | if ( Test ) Stmt else Stmt
While   ::= while ( Test ) Stmt
```

Solution: look at input tokens to help decide

Predictive parsing

Predictive parser:
top-down parser that can select correct rhs looking at at most *k* input tokens (the **lookahead**)

Efficient:

- no backtracking needed
- linear time to parse

Implementation of predictive parsers:

- recursive-descent parser
  - each nonterminal parsed by a procedure
  - call other procedures to parse sub-nonterminals, recursively
  - typically written by hand
- table-driven parser
  - PDA: like table-driven FSA, plus stack to do recursive FSA calls
  - typically generated by a tool from a grammar specification

LL(*k*) grammars

Can construct predictive parser automatically/easily if grammar is **LL(*k*)**

- **L**eft-to-right scan of input, **L**eftmost derivation
- ***k*** tokens of lookahead needed, ≥ 1

Some restrictions:

- no ambiguity (true for any parsing algorithm)
- no **common prefixes** of length ≥ *k*:

```
If ::= if ( Test ) Stmt end
    | if ( Test ) Stmt else Stmt end
```

- no **left recursion**:

```
E ::= E + T | T
```

- a few others

Restrictions guarantee that, given *k* input tokens, can always select correct rhs to expand nonterminal

Eliminating common prefixes

Can **left factor** common prefixes to eliminate them

- create new nonterminal for different suffixes
- delay choice till after common prefix

Before:

```
If     ::= if ( Test ) Stmt end
         | if ( Test ) Stmt else Stmt end
```

After:

```
If     ::= if ( Test ) Stmt IfCont
IfCont ::= end
         | else Stmt end
```

Grammar a bit uglier

Easy to do by hand in recursive-descent parser

Eliminating left recursion

Can rewrite grammar to eliminate left recursion

Before:

```
E ::= E + T | T
T ::= T * F | F
F ::= id | ...
```

After:

```
E     ::= T ECont
ECont ::= + T ECont | ε
T     ::= F TCont
TCont ::= * F TCont | ε
F     ::= id | ...
```

After, in sugared form:

```
E ::= T { + T }
T ::= F { * F }
F ::= id | ...
```

Sugared form pretty readable still

Easy to implement in hand-written recursive descent

Grammar no longer specifies associativity; must add meta-rules