COS 301
Programming Languages

Syntax & Semantics

Syntax & semantics

- **Syntax**:
  - Defines correctly-formed components of language
  - *Structure* of expressions, statements
- **Semantics**: *meaning* of components
- Together: define the programming language

Simplicity:

A language that is simple to parse for the compiler is also simple to parse for the human programmer.

N. Wirth

Simple to parse?

```
sub b{$n=99-@_-$_||No;"$n
  bottle"."s"x!!--$n." of beer"};$w="
  on the wall"; die map{b."$w,\n".b.",
  \nTake one down, pass it around,
  \n".b(0)."$w.\n\n"}0..98;
```

Describing syntax

- Not sufficient for PL to have syntax
- Have to be able to describe it to
  - programmers
  - implementers (e.g., compiler designers)
  - automated compiler generators, verification tools
- *Specification*:
  - Humans: some ambiguity okay
  - Automated tools: must be unambiguous
  - For programmers: unambiguous >> ambiguous!

Terminology

- Alphabet:
  - a set of characters
  - small (e.g., {0,1}, {A-Z}) to large (e.g., Kanji)
- Sentence:
  - string of characters drawn from alphabet
  - conforms to syntax rules of language
- Language: set of sentences
- Lexeme (token):
  - smallest syntactic unit of language
  - e.g., English: words
  - e.g., PL: 1.0, `*`, `sum`, `begin`, …
  - Token type: *category* of lexeme (e.g., identifier)

Tokens & lexemes

- "Lexeme" often use interchangeably with "token"
- Example:

```
index = 2 * count + x
```

Lexeme	Token type	Value
index	identifier	"index"
=	assignment	
2	int	literal 2
count	identifier	"count"
+	addition	
17	int	literal 17

Lexical rules

- Lexical rules: define set of legal lexemes
- Lexical, syntactical rules specified separately
  - Different types of grammars
  - Recognized differently
    - different kinds of *automata*
    - different parts of compiler/interpreter
- Lexical rules: *regular expressions*
- ⇒ their grammar = *regular grammars*
- Parsed by *finite automata* (finite state machines)

Formal Languages

Formal languages

- Defined by *recognizers* and *generators*
- *Recognizers:*
  - reads input strings over alphabet of language
  - decides: is string sentence in the language?
  - Ex.: *syntax analyzer* of compiler
- *Generators:*
  - Generates sentences in the language
  - Determine if string ∈ of {sentences}: compare to generator's structure
  - Ex: a grammar

Recognizers & generators

- Recognizers and generators: closely related
- Given grammar (generator), we can ⇒ recognizer (parser)
- Oldest system to do this:
  - yacc (Yet Another Compiler Compiler)
  - still widespread use
  - GNU bison

Chomsky Hierarchy

- Formal language hierarchy – Chomsky, late 50s
- Four levels:
  - Regular languages
  - Context-free languages
  - Context-sensitive languages
  - Recursively-enumerable languages (unrestricted)
- Only regular and context-free grammars in PL

Context-free grammars

- Regular grammars: not powerful enough to express PLs
- Context-free grammars (CFGs):
  - sufficient
  - relatively easy to parse
- Need way to specify context-free grammars
- Most common way: Backus-Naur Form

BNF

- John Backus [1959]; extended by Peter Naur
- Created to describe Algol 60
- Any context-free grammar can be written in BNF
- Apparently similar to 2000 year-old notation for describing Sanskrit!

BNF

- BNF is a *metalanguage*
- Symbols represent syntactic structures: `<assign>`, `<ident>`, etc.
- *Non-terminals* & *terminal* symbols
- *Productions*:
  - *Rewrite rules*: show how one pattern ⇒ another
  - Context-free languages: production shows how non-terminal ⇒ sequence of non-terminals, terminals

```
<assign> → <var> = <expression>
```

  - LHS/antecedent, RHS/consequent

BNF formalism

- A grammar for a PL is a set: {P,T,N,S}
  - T = set of *terminal symbols*
  - N = set of *non-terminal symbols* ($T \cap N = \{\}$)
  - S = start symbol ($S \in N$)
  - P = set of *productions:*

$$A \rightarrow \omega$$

where $A \in N$ and $\omega \in (N \cup T)^*$

set of all strings of terminals and non-terminals

BNF

- *Sentential form*: string of symbols
- Productions:
  - S → S’
  - S, S’ are sentential forms
- Nonterminal symbols N:
  - *grammatical categories*
  - E.g., identifier, expression, program
- Designated start symbol S: often `<program>`
- Terminal symbols T: lexemes/tokens

BNF symbols

- Nonterminals: written in angle brackets or in special font: `<expression>`
- Can have ≥ 1 rule/nonterminal — write as one rule
- Alternatives: specified by | - e.g.,

```
<stmt> → <single_stmt> |
         begin <stmt_list> end
```

or

```
<stmt> ::= <single_stmt> |
           begin <stmt_list> end
```


Recursion in BNF

- *Recursion:* lets finite grammar ⇒ infinite language
- Direct recursion:
  - LHS appears on the RHS
  - E.g., specify a list:

```
<ident_list> ::= ident |
                 ident, <ident_list>
```

- Indirect recursion:

```
<expr> ::= <expr> + <term> | ...
<term> ::= <factor> | ...
<factor> ::= (<expr>) | ...
```

Derivations

- Let s be a *sentence* produced by a grammar G
- A *language* L defined by grammar G:

  L = {s | G produces s from S}

  - Recall: Sentence composed only of terminal symbols
  - Produced in 0 or more steps from G's start symbol S
- **Derivation** of sentence s = list of rules

$$S \xRightarrow{r_1} s_1 \xRightarrow{r_2} s_2 \xRightarrow{r_3} \cdots \xRightarrow{r_k} s$$

i.e., $r_1, r_2, r_3, \ldots r_k$

An Example Grammar

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```


An Example Derivation

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

```
<program> ⟹ <stmts>
          ⟹ <stmt>
          ⟹ <var> = <expr>
          ⟹ a = <expr>
          ⟹ a = <term> + <term>
          ⟹ a = <var> + <term>
          ⟹ a = b + <term>
          ⟹ a = b + const
```

Derivations

- Every string in a derivation: *sentential form*
- Derivations can be *leftmost* or *rightmost*
- **Leftmost derivation:** leftmost nonterminal in each sentential form is expanded first

Example

- Given G = { T, N, P, S }
  - T = { a, b, c }
  - N = { A, B, C, W }
  - S = { W }
- Is string *cbab* ∈ L(G)? I.e., ∃ derivation D from start S to *cbab?*
- P =

1. W → AB	or	<W> ::= <A><B>
2. A → Ca		<A> ::= <C>a
3. B → Ba		<B> ::= <B>a
4. B → Cb		<B> ::= <C>b
5. B → b		<B> ::= b
6. C → cb		<C> ::= cb
7. C → b		<C> ::= b

Leftmost derivation

Begin with the start symbol W and apply production rules expanding the leftmost non-terminal.

1. W → AB
2. A → Ca
3. B → Ba
4. B → Cb
5. B → b
6. C → cb

W ⟹ AB	Rule 1. W → AB
AB ⟹ CaB	Rule 2. A → Ca
CaB ⟹ *cba*B	Rule 6. C → cb
*cba*B ⟹ *cbab*	Rule 5. B → b

∴ cbab ∈ L(G)

Rightmost derivation

Begin with the start symbol W and apply production rules expanding the rightmost non-terminal.

1. W → AB
2. A → Ca
3. B → Ba
4. B → Cb
5. B → b
6. C → cb

W → AB	Rule 1. W → AB
AB → Ab	Rule 5. B → b
Ab → Cab	Rule 2. A → Ca
Cab → *cbab*	Rule 6. C → cb

∴ cbab ∈ L(G)

Rightmost derivation: 1 → 5 → 2 → 6

Shorter version of G

Using selection (options) in the RHS

G = { T, N, P, S }
T = { a, b, c }
N = { A, B, C, W }
S = { W }

1. W → AB
2. A → Ca
3. B → Ba
4. B → Cb
5. B → b
6. C → cb

1. W → AB or <W> ::= <A><B>
2. A → Ca <A> ::= <C>a
3. B → Ba | Cb | b <B> ::= <B>a | <C>b | b
4. C → cb | b <C> ::= cb | b

Your Turn!

G = {T,N,P,S}
T = { a, b, c }
N = { A, B, C, W }
S = { W }
P =

1. W → AB <W> ::= <A><B>
2. A → Ca <A> ::= <C>a
3. B → Ba | Cb | b <B> ::= <B>a | <C>b | b
4. C → cb | b <C> ::= cb | b

1. Is cbbacbb in L?
2. Is baba in L?
3. Show a leftmost derivation for cbabb
4. Show a rightmost derivation for cbabb

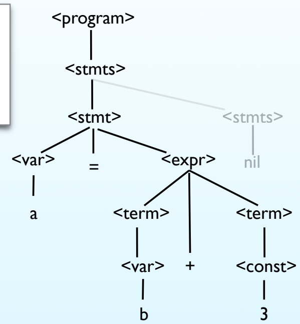
Derivations as parse trees

- **Parse tree**: graphical representation of a derivation
- *Root*: the start symbol
- Each *node* + *children* = rule application
  - LHS = node
  - RHS = children
- *Leaves*: terminal symbols in derived sentence

Parse tree

`a = b + 3`

```
<program> ::= <stmts>
<stmts>   ::= <stmt> <stmts>
            | nil
<stmt>    ::= <var> = <expr>
<var>     ::= a | b | …
<const>   ::= number
<expr>    ::= <term> + <term>
```



Example grammar: Assignment

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <id> + <expr> |
             <id> * <expr> |
             ( <expr> )    |
             <id>
```

Example derivation

```
A = B * ( A  +  C )
<assign>  ⟹ <id> = <expr>
          ⟹ A = <expr>
          ⟹ A = <id> * <expr>
          ⟹ A = B * <expr>
          ⟹ A = B * ( <expr> )
          ⟹ A = B * ( <id> + <expr> )
          ⟹ A = B * ( A + <expr> )
          ⟹ A = B * ( A + <id> )
          ⟹ A = B * ( A + C )
```

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <id> + <expr> |
             <id> * <expr> |
             ( <expr> )    |
             <id>
```


Ambiguity

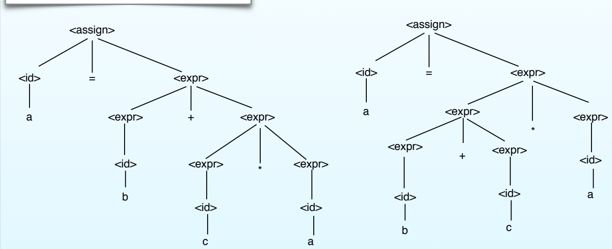
Ambiguous grammar if sentential form ⟹ ≥ 1 parse tree

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <expr> + <expr>
           | <expr> * <expr>
           | ( <expr> )
           | <id>
```


Ambiguity

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <expr> + <expr>
           | <expr> * <expr>
           | ( <expr> )
           | <id>
```

a = b + c * a



a = b + (c * a)

a = (b + c) * a

What causes ambiguity?

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <id> + <expr>
           |  <id> * <expr>
           | ( <expr> )
           | <id>
```

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <expr> + <expr>
           | <expr> * <expr>
           | ( <expr> )
           | <id>
```

- Example *unambiguous* grammar:
  - <expr> allowed to grow only on right
- Example *ambiguous* grammar:
  - <expr> can be expanded right or left
- General case: *Undecidable* whether grammar is ambiguous
- Parsers: use “extra-grammatical” information to disambiguate

Ambiguity

- How do *we* avoid ambiguity when evaluating (say) arithmetic expressions?
- E.g.: 5 + 7 * 3 + 8 ** 2 ** 3
- Precedence
- Associativity

Precedence

- Want grammar to enforce precedence
- Code generation follows parse tree structure
- For a parse tree:
  - To evaluate node, all children must be evaluated
  - ⟹ things lower in tree evaluated first
  - ⟹ things lower in tree have higher precedence
- So: write grammar to generate this kind of parse tree

Precedence in grammars

- Example: grammar with no precedence
  - generates tree where rightmost operator is lower:

```
<assign> ::= <id> = <expr>
    <id> ::= A | B | C
  <expr> ::= <id> + <expr>
           | <id> * <expr>
           | (<expr>)
           | <id>
```

- In A + B * C: multiplication will be first
- In A * B + C: addition will be first

Enforcing precedence

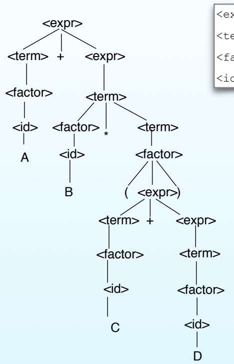
- Higher-precedence operators → lower in tree
  - ensure derivation → higher-precedence operators is longer than → lower-precedence
  - ⇒ create new *category* for each precedence level
  - Make higher-order categories/levels appear deeper
- E.g.: instead of just `<expr>` and `<id>`, have:
  - `<expr>` – entire (sub)expressions; precedence level of plus/minus
  - `<term>` – multiplication/division precedence
  - `<factor>` – parentheses/single <id> precedence
  - `<id>` – represent identifiers

A grammar with precedence

```
<expr>   ::=  <term> + <expr>
            | <term> - <expr> | <term>
<term>   ::=  <term> * <factor>
            | <term> / <factor> | <factor>
<factor> ::= ( <expr> )
            | <id>
<id>     ::= A | B | C | D
```

Example

A+B*(C+D)




```
<expr>   ::=  <term> + <expr>
              | <term> - <expr> | <term>
<term>   ::=  <term> * <factor>
              | <term> / <factor> | <factor>
<factor> ::= ( <expr> )
              | <id>
<id>     ::= A | B | C | D
```


Associativity

- *Associativity*: order to evaluate operators at same level
- E.g.:
  - Left-to-right:

    `5 - 4 - 3 = (5 - 4) - 3 = 1 - 3 = -2`

    What if it were R→L?
  - Right-to-left:

    `2**3**2 = 2**(3**2)= 2**9 = 512`

    What if it were L→R?

Associativity

- Previous example grammar: left-associative

```
<term> ::= <term> * <factor> | …
```

- Right associativity:
  - reverse where recursion occurs
  - may need to introduce new category

```
 <factor> ::= <primary> ** <factor>
              | <primary>
<primary> ::= <id> | ( <expr> )
```

Precedence/associativity (summary)

- Precedence:
  - determined by length of shortest derivation from start → operator
  - shorter derivations ⇒ lower precedence
- Associativity: determined using left or right recursion

Your turn

- Given
  - Factorial has higher priority than exponentiation
  - Assignment is right-associative
- How would you change this grammar to handle both?

```
<expr>    ::=  <term> + <expr>
             | <term> - <expr> | <term>
<term>    ::=  <term> * <factor>
             | <term> / <factor> | <factor>
<factor>  ::= <primary> ** <factor>
             | <primary>
<primary> ::= <id> | ( <expr> )
<id>      ::= A | B | C | D
```


Problems

- Some languages have *too* many precedence levels
- E.g., C++:

Precedence	Operator	Description	Example	Associativity
1	::	Scoping operator	Class::age = 2;	none

2	()
	[]
	->
	.
	++
	--
		Grouping operator
		Array access
		Member access from a pointer
		Member access from an object
		Post-increment
		Post-decrement
			(a + b) / 4;
			array[4] = 2;
			ptr->age = 34;
			obj.age = 34;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
				left to right

3	!
	~
	++
	--
	-
	+
	*
	&
	(type)
	sizeof
		Logical negation
		Bitwise complement
		Pre-increment
		Pre-decrement
		Unary minus
		Unary plus
		Dereference
		Address of
		Cast to a given type
		Return size in bytes
			if(!done) ...
			flags = ~flags;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			int i = -1;
			int i = +1;
			data = *ptr;
			address = &obj;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				right to left

4	->* .*	Member pointer selector Member object selector	ptr->*var = 24; obj.*var = 24;	left to right

5	*
	/
	%
		Multiplication
		Division
		Modulus
			int i = 2 * 4;
			float f = 10 / 3;
			int rem = 4 % 3;
				left to right

Problems

6	+	Addition	int i = 2 + 3;	left to right
	-	Subtraction	int i = 5 - 1;	
7	<<	Bitwise shift left	int flags = 33 << 1;	left to right
	>>	Bitwise shift right	int flags = 33 >> 1;	
8	<	Comparison less-than	if(i < 42) ...	left to right
	<=	Comparison less-than-or-equal-to	if(i <= 42) ...	
	>	Comparison greater-than	if(i > 42) ...	
	>=	Comparison geater-than-or-equal-to	if(i >= 42) ...	
9	==	Comparison equal-to	if(i == 42) ...	left to right
	!=	Comparison not-equal-to	if(i != 42) ...	
10	&	Bitwise AND	flags = flags & 42;	left to right
11	^	Bitwise exclusive OR	flags = flags ^ 42;	left to right
12	|	Bitwise inclusive (normal) OR	flags = flags | 42;	left to right

Problems

13	&&	Logical AND	if(conditionA && conditionB) ...	left to right
14	||	Logical OR	if(conditionA || conditionB) ...	left to right
15	? :	Ternary conditional (if-then-else)	int i = (a > b) ? a : b;	right to left
16	=	Assignment operator	int a = b;	right to left
	+=	Increment and assign	a += 3;	
	-=	Decrement and assign	b -= 4;	
	*=	Multiply and assign	a *= 5;	
	/=	Divide and assign	a /= 2;	
	%=	Modulo and assign	a %= 3;	
	&=	Bitwise AND and assign	flags &= new_flags;	
	^=	Bitwise exclusive OR and assign	flags ^= new_flags;	
	|=	Bitwise inclusive (normal) OR and assign	flags |= new_flags;	
	<<=	Bitwise shift left and assign	flags <<= 2;	
	>>=	Bitwise shift right and assign	flags >>= 2;	
17	,	Sequential evaluation operator	for(i = 0, j = 0; i < 10; i++, j++) ...	left to right

Design choices

- Lots of precedence levels → complicated
  - Readability decreased
  - E.g.,
    - C++ has 17 precedence levels
    - Java has 16
    - C has 15
  - In all three: some operators left-, some right-associative
- Avoid too few or odd choices
  - E.g., Pascal (5 levels)

`A <= 0 or 100 <= 0` Error: “or” > “<=”

Should be:

`(A <= 0) or (100 <= 0)`

Design choices

- Avoid too few or odd choices (cont'd):
  - APL:
    - No precedence at all!
    - All operators are right-associative
  - Smalltalk:
    - Technically no "operators" per se
    - Operators are binary messages
    - E.g., 3 + 20 / 5:
      - First: "+" message to object "3", arg. "20" ⇒ object "23"
      - Then "/" message to "23", arg. "5" ⇒ object "4.6"
    - ⇒ As if no precedence, everything left-associative
    - Meaning depends on receiving class' implementation
- …Or, make sure it's completely clear:

  Lisp: (+ 3 (/ 20 5)) Forth: 3 20 5 / +

Complexity of grammars

- C++: large number of operators, precedence levels
- Each precedence level ⇒ new non-terminal (category)
- Grammar ⇒ large, difficult to read
- Instead of large grammar:
  - Write small, ambiguous grammar
  - Specify precedences, associativity *outside* the grammar

Example grammar: A small, C-like language

```
Expression → Conjunction { || Conjunction }
Conjunction → Equality { && Equality }
Equality → Relation [ EquOp Relation ]
EquOp → == | !=
Relation → Addition [ RelOp Addition ]
RelOp →  < | <= | > | >=
Addition → Term { AddOp Term }
AddOp → + | -
Term → Factor { MulOp Factor }
MulOp →  * | / | %
Factor → [ UnaryOp ] Primary
UnaryOp →  - | !
Primary → Identifier [  [ Expression ] ] | Literal
         |  ( Expression )  | Type ( Expression )
```

Syntax and semantics

- Parse trees embody the syntax of a sentence
- Should also correspond to *semantics* of sentence
  - precedence
  - associativity
- Extends beyond expressions
  - e.g., the "dangling else" problem

Dangling else

```
<IfStatement> ::=   if ( <Expression> ) <Statement>
                  | if ( <Expression> ) <Statement>
                    else <Statement>

<Statement> ::= <Assignment>
              | <IfStatement>
              | <Block>

<Block> ::= { <Statements> }

<Statements> ::= <Statements> <Statement>
               | <Statement>
```

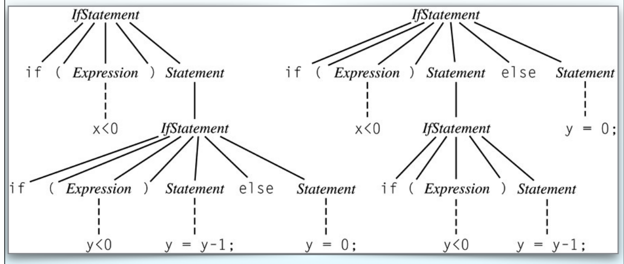

Dangling else

- Problem: which "if" does the "else" belong to (associate with)?

```
if (x < 0)
    if (y < 0) y = y - 1;
    else y = 0;
```

- Answer: either one!

Parse trees for the statement



Solution?

- Conventions (maybe extra-grammatical):
  - Associate each **`else`** with *closest* **`if`**
  - Use **`{}`** or **`begin/end`** to override
  - E.g., Algol 60, C, C++, Pascal
- Explicit delimiters:
  - Begin, end every conditional: {}, if…fi, begin…end, indentation level
  - Algol 68, Modula, Ada, VB, Python
- Rewrite grammar to limit what can appear in conditional:

```
<IfThenStatement> ::= if ( <Expression> ) <statement>
<IfThenElseStatement> ::= if ( <Expression> ) <StatementNoShortIf>
                          else <Statement>
```

where `<StatementNoShortIf>` – everything except `<IfThenStatement>`

Extended BNF

Audiences

- Grammar specification language: means of communicating to *audience*
  - Programmers: What do legal programs look like?
  - Implementers: need exact, detailed definition
  - Tools (e.g., parsers/scanner generators): need exact, detailed definition in machine-readable form
- Maybe use more readable specification for humans
  - Needs to be unambiguous
  - Must be able to ⇒ machine-readable form (e.g., BNF)

Extended BNF

- BNF developed in late 1950s — still widely used
- Original BNF — a few minor inconveniences — e.g.:
  - recursion instead of iteration
  - verbose selection syntax
- *Extended BNF* (EBNF): increases readability, writability
  - Expressive power unchanged: still CFGs
  - Several variations

EBNF: Optional parts

- Brackets [] delimit optional parts

```
<proc_call> → ident ([<expr_list>])
```

- Instead of:

```
<proc_call> → ident()
            | ident (<expr_list>)
```

EBNF: Alternatives

- Specify *alternatives* in (), separated by "|"

```
<term> → <term> (+|-) factor
```

- Replaces

```
<term> →  <term>  + factor
        | <term>  - factor
```

- So what about replacing:

```
<term> → <term> + <factor> | <term> - <factor>
                | <factor>
```

⟹

```
<term> → (<term> (+|-) <factor> | <factor>)
```

or

```
<term> → [<term> (+|-) ] <factor>
```


EBNF: Recursion

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

- Replaces

```
<ident> → letter
        | <ident> letter
        | <ident> digit
```


BNF and EBNF

- BNF

```
<expr> →   <expr> + <term>
         | <expr> - <term>
         | <term>
<term> →   <term> * <factor>
         | <term> / <factor>
         | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

EBNF: Associativity

- Note that the production:

```
<expr> → <term> { ( + | - ) <term> }
```

does not seem to specify the left associativity that we have in

```
<expr> → <expr> + <term>
       | <expr> + <term> | <term>
```

- In EBNF left associativity is usually assumed
  - Enforced by EBNF-based parsers
  - Explicit recursion used for right associative operators
  - Some EBNF grammars may specify associativity outside of the grammar

EBNF variants

- Alternative RHSs are put on separate lines
- Use of a colon instead of "→"
- Use of $_{\texttt{opt}}$ for optional parts
- Use of `oneof` for choices

EBNF to BNF

- Can always rewrite EBNF grammar as BNF grammar — e.g.:

```
<A> → x { y } z
```

- can be rewritten:

```
<A>  → x <A1> z
<A1> → ε | y <A1>
```

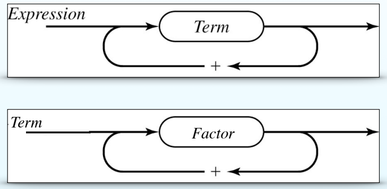
- where ε is a standard symbol *empty string* (sometimes λ)
- Rewriting EBNF rules with ( ), [ ] — done similarly
- EBNF is no more powerful than BNF…
- …but rules often simpler and clearer for human readers

Syntax Diagrams

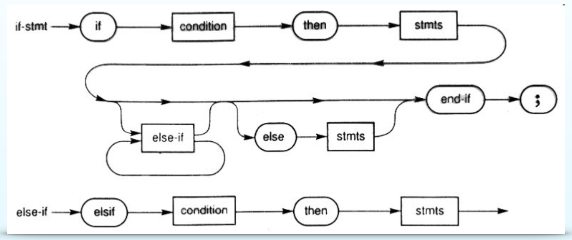
Syntax Diagrams

- Similar goals as EBNF — aimed at humans, not machines
- Introduced by Jensen and Wirth with Pascal in 1975
- Pictorial rather than textual

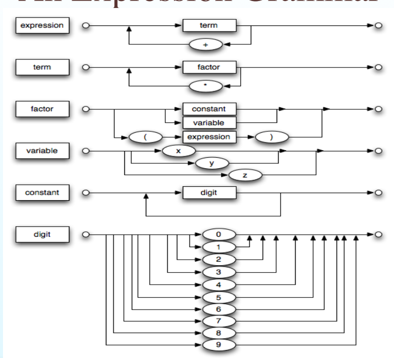
Ex: Expressions with addition



A More Complex Example



An Expression Grammar



From http://en.wikipedia.org/wiki/Syntax_diagram

Static Semantics

Problem with CF grammar for PLs

- Some aspects of PL — not easily express in CFG
- E.g.:
  - Assignment statement LHS' type must be compatible with RHS'
    - type of LHS has to match type of RHS
    - *could* be done in CFG…
    - …but cumbersome
  - All variables have to be declared before used
    - *cannot* be expressed in BNF

Static semantics

- These kinds of constraints: *static semantics*
  - Only indirectly related to *meaning*
  - Helps define program's legal form (syntax)
  - Most rules: typing
  - Can be done at *compile time* (⇒ static)
- *Dynamic semantics* – runtime behavior/meaning of program

Attribute grammars

- AG [Knuth, 1968] used in addition to CFG
- Let's parse tree nodes carry some semantic info
- AG is CFG + :
  - *attributes*:
    - associated with terminals & non-terminals
    - similar to variables – values can be assigned
  - *attribute computation (semantic) functions*
    - assoc. with grammar rules
    - say how attribute values are computed
  - *predicate functions*
    - state semantic rules
    - assoc. with grammar rules

Definition

- Attribute grammar G = context-free grammar &:
  - Each grammar symbol x in N has a set $A(x)$ of attribute values
    - $A(x)$ consists of two disjoint sets:
      - $S(x)$ and $I(x)$, the
      - *Synthesized attributes* $S(x)$
      - *Inherited attributes* $I(x)$
  - Each rule $r \in$ P has
    - set of functions ⇒ each defines certain attributes of rule's nonterminals
    - set of predicates ⇒ check for attribute consistency

Intrinsic attributes

- *Intrinsic attributes* – values determined outside the parse tree
- Attributes of leaf nodes
- Ex: Type of a variable
  - Obtained from *symbol table*
  - Value from declaration statements
- Initially: the only attributes are intrinsic
- Semantic functions compute the rest

Synthesized attributes

- "Synthesized" = "computed"
- Means of passing semantic information up parse tree
- Synthesized attributes for grammar rule:

  $X_0 \rightarrow X_1 \ldots X_n$

  for $S(X_0) = f(A(X_1)...A(X_n))$ ⇐ attribute function
- Value of synthesized attributes depends only on value of children attributes
- E.g.: an "actual type" attribute of a node
  - For variable: declared type
  - For constant: defined
  - For expression: *computed* from type of parts

Inherited attributes

- Pass semantic information down, across parse tree
- Attributes of child ⇐ parent
- For a grammar rule
  $X_0 \rightarrow X_1 \ldots X_j \ldots X_n$
  inherited attributes $S(X_j) = f(A(X_0),\ldots,A(X_{j-1}))$
- Value depends only on attributes of parent, siblings (usually left siblings)
- E.g.: "expected type" of expression on RHS of assignment statement ⇐ type of variable on LHS
- E.g.: "type" in a type declaration ⇒ identifiers

Predicate functions

- *Predicates* = Boolean expressions on
  $\bigcup_i A(X_i)$
  and a set of literal values (e.g., `int`, `float`,…)
- Valid derivation iff every nonterminal's predicate true
- Predicate false ⇒ rule violation ⇒ ungrammatical

Attributed/decorated parse trees

- Each node in parse tree has (possibly empty) set of attributes
- When all attributes computed, tree is *fully attributed (decorated)*
- Conceptually, parse tree could be produced, then decorated

Example

- In Ada, the `end` of a procedure has specify the procedure's name:

```
procedure simpleProc …
…
end simpleProc;
```

- Can't do this in BNF!
- Syntax rule:

```
<proc_def> → procedure <proc_name>[1]
                <proc_body> end <proc_name>[2]
```

- Predicate:

```
<proc_name>[1].string == <proc_name>[2].string
```


Example 2 (from book)

An attribute grammar for simple assignment statements

```
1. Syntax rule:  <assign> → <var> = <expr>
   Semantic rule:
       <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule:
    <expr>.actual_type ←
           if (<var>[2].actual_type = int) &
              (<var>[3].actual_type = int)
           then int
           else real
 Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule:  <expr>.actual_type ←
<var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ←
                                     look-up(<var>.string)
```

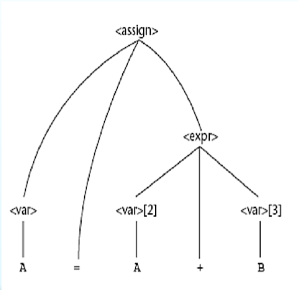
where "look-up(n)" looks up a name in the symbol table and returns its type

Example 2

- `actual_type` – synthesized attribute
  - computed sometimes
  - also intrinsic for `<var>`
- `expected_type` - inherited attribute
  - computed in this example
  - but associated with nonterminal

Example – parse tree

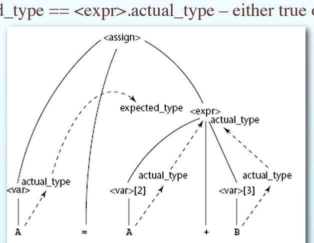
`A = A + B`



- Computing attribute values
  - Could be top-down, if all inherited
  - Could be bottom-up, if all synthesized
  - Mostly mixed
- General case: need dependency graph to determine evaluation order

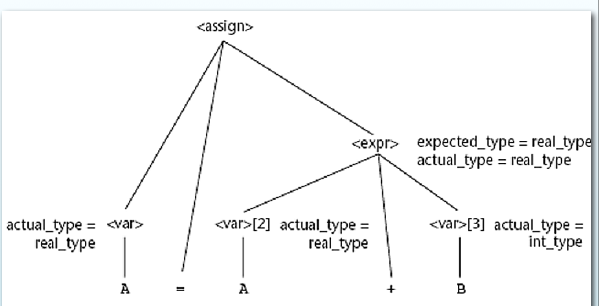
Decorating the tree

1. <var>.actual_type ← lookup(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← lookup(A) (Rule 4)
4. <var>[3].actual_type ← lookup(B) (Rule 4)
5. <expr>.actual_type ← (int | real) (Rule 2)
6. <expr>.expected_type == <expr>.actual_type – either true or false (Rule 2)



Decorated tree

Assume `A` is `real`, `B` is `int`

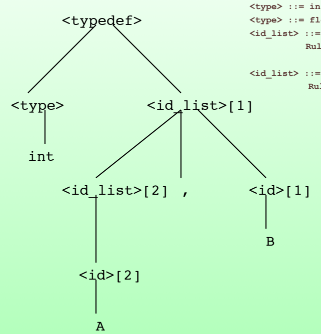


Example 3: inherited

```
<typedef> ::= <type> <id_list>
    Rule: <id_list>.type ← <type>.type
<type> ::= int
    Rule: <type>.type ← int
<type> ::= float
    Rule: <type>.type ← float
<id_list> ::= <id_list>_1 , <id>
    Rules: <id_list>_1.type ← <id_list>.type
           <id>.type ← <id_list>.type
<id_list> ::= <id>
    Rule: <id>.type ← <id_list>.type
```


Parse tree

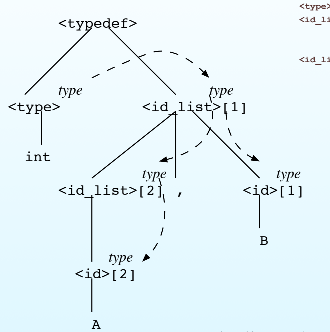
int A, B




```
<typedef> ::= <type> <id_list>
    Rule: <id_list>.type ← <type>.type
<type> ::= int    Rule: <type>.type ← int
<type> ::= float    Rule: <type>.type ← float
<id_list> ::= <id_list>_1 , <id>
    Rules: <id_list>_1.type ← <id_list>.type
           <id>.type ← <id_list>.type
<id_list> ::= <id>
    Rule: <id>.type ← <id_list>.type
```


Evaluation order

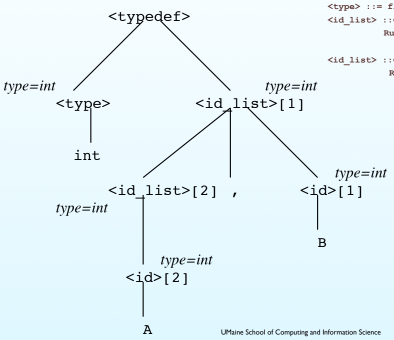
int A, B




```
<typedef> ::= <type> <id_list>
    Rule: <id_list>.type ← <type>.type
<type> ::= int    Rule: <type>.type ← int
<type> ::= float    Rule: <type>.type ← float
<id_list> ::= <id_list>_1 , <id>
    Rules: <id_list>_1.type ← <id_list>.type
           <id>.type ← <id_list>.type
<id_list> ::= <id>
    Rule: <id>.type ← <id_list>.type
```

Decorated tree

```
int A, B
```




```
<typedef> ::= <type> <id_list>
          Rule: <id_list>.type ← <type>.type
<type> ::= int    Rule: <type>.type ← int
<type> ::= float    Rule: <type>.type ← float
<id_list> ::= <id_list>_1 , <id>
          Rules: <id_list>_1.type ← <id_list>.type
                 <id>.type ← <id_list>.type
<id_list> ::= <id>
           Rule: <id>.type ← <id_list>.type
```


Dynamic Semantics

Dynamic semantics

- Static semantics – still about syntax
- *Dynamic semantics:* describes the <u>meaning</u> of statements, program
- Why is it needed?
  - Programmers: need to know what statements mean
  - Compiler writers:
    - compiler has to produce semantically-correct code
    - also for compiler generators (yacc, bison)
  - Automated verification tools: correctness proofs
  - Designers: find ambiguities, inconsistencies
  - Ways of reasoning about semantics: Operational, denotation, axiomatic

Operational Semantics

Operational semantics

- *Operational semantics:*

  meaning = statement's *effects* on a machine
- *Machine*: real or mathematical
- Machine state: contents of memory, registers, PC, etc.
- *Effects* = changes in state
- You've probably used this informally:
  - write down variables, values
  - walk through code, tracking changes
- Problems:
  - Changes in real machine state too small, too numerous
  - Storage too large & complex

Operational semantics

- Need:
  - *intermediate language* — coarser state
  - *virtual machine:* interpreter for idealized computer
- Ex: programming texts
  - Define a construct in terms of simpler operations
  - E.g., C loop as conditionals + goto
- Your book:

```
ident = var bin_op var
ident = unary_op var
goto label
if var relop var goto label
```

This can describe semantics of most loop constructs

Operational Semantics

E.g., a **while** loop:

```
      ident = var
head  if var relop var goto end
      <statements>
      goto head
end   …
```

E.g., C's `for` loop:

```
for (e1;e2;e3) stmt;

        e1
loop:   if e3 == 0 goto end
        stmt
        e2
        goto loop
end:    …
```


Operational semantics

- Good for textbooks and manuals, etc.
- Used to describe semantics of PL/I
- Works for simple semantics – not usually the case (certainly not for PL/I)
- Relies on reformulating in terms of simpler PL, not math…
- …can $\Longrightarrow$ imprecise semantics, circularities, interpretation differences
- Better: use *mathematics* to describe semantics

Denotational Semantics

Denotational semantics

- Scott & Strachey (1970)
- Based on *recursive function theory*
- Define mathematical object for each language entity
- *Mapping function:*

  Language entities → mathematical objects

  - Domain = syntactic domain
  - Range = semantic domain

Denotational semantics

- Meaning of constructs: defined *only* by value of program's variables:
  - state `s = {<i1,v1>, <i2,v2>,…}`
  - `VARMAP(ij,s)`
- *Statement* – defined as *state-transforming function*
- *Program* – collection of functions operating on state

Denotational semantics: Binary numbers

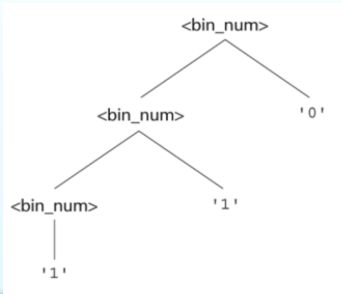
- Grammar:

$$
\begin{aligned}
< binNum > \; &\rightarrow \; '0' \\
&| \; '1' \\
&| \; < binNum >' 0' \\
&| \; < binNum >' 1'
\end{aligned}
$$

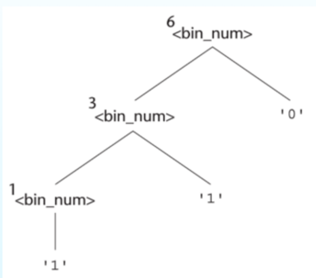
- Let M_{bin} be mapping function

$$
\begin{aligned}
&M_{bin}('0') = 0 \\
&M_{bin}('1') = 1 \\
&M_{bin}(< binNum > \; '0') = 2 \times M_{bin}(< binNum >) \\
&M_{bin}(< binNum > \; '1') = 2 \times M_{bin}(< binNum >) + 1
\end{aligned}
$$

Denotational semantics: Binary numbers



Denotational semantics: Binary numbers



Denotational semantics: Expressions

- Assume only:
  - numbers drawn from $\mathbb{Z}$ (integers)
  - variables
  - binary expressions with two subexpressions and an operator
- Map an expression onto $\mathbb{Z} \cup \{\text{error}\}$

Denotational semantics: Loops

- Meaning of a loop = value of variables after the loop has executed the correct number of times (assuming no errors)
- Loop is converted from iteration to *recursion*
- Recursive control is mathematically defined by other recursive state mapping functions
- Recursion is easier to describe mathematically than iteration

Den. semantics: pretest loop

M_l(while B do L, s) Δ=
```
if Mb(B, s) == undef
    then error
    else if Mb(B, s) == false
        then s
        else if Msl(L, s) == error
              then error
              else Ml(while B do L, Msl(L, s))
```


Using denotational semantics

- Can prove *correctness* of programs
- Rigorous way to think about programs
- Can aid language design
- But: due to complexity, of little use to most language users

Axiomatic Semantics

Axiomatic semantics

- Based on formal logic (predicate calculus)
- Specifies what can be proven about the program — not meaning per se
- Can be used for *program verification*
- *No* model of machine state, program state, or state changes
- Instead: meaning based on relationships between variables and constants – same for every execution
- *Axioms* (assertions) defined for each statement type
  - What is true before and after the statement with respect to program variables
  - This defines the semantics of the statement

Assertions

- *Preconditions*: What is true (constraints on the program variables) before a statement
- *Postconditions*: What is true after the statement executes
- Postcondition of one statement becomes precondition of next
- Start with postcondition of program itself (last statement)
- Go backward to preconditions obtaining at program start ⇒ program is correct

Assertions

- Example:

  `{P} x = cos(y) {x > 0}`

- What is precondition P?
- Possibilities:

  $\{0 \le y < 90\}$, $\{10 \le y \le 80\}$, $\{-90 < y < 90\}$...

- Which to choose?
- Choose *weakest precondition*
  - Sometimes can be specified by axiom
  - Usually only by *inference rule*

Axiomatic semantics for assignment

- Given `v = E` with postcondition Q:
  - Precondition P is computed by replacing all instances of v with E in Q
  - Ex:

```
y = 2x + 7, Q = {y > 3}
2x + 7 > 3
2x > -4
x > -2 = P
```

- Usually written as:

  $\{Q_{x \to E}\}$ `x = E {Q}`

  `e.g.: {x > -2} y = 2x + 7 {y > 3}`

Axiomatic semantics: if-then-else

- Sometimes, need more than an axiom – need an *inference rule* to specify semantics
- Inference rule has form:

$$\frac{S_1, S_2, \ldots, S_n}{S}$$

- Inference rule for if-then-else:

$$\frac{\{B \wedge P\}\ S_1\ \{Q\}, \{\neg B \wedge P\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

- $\Rightarrow$ Have to prove case both when B is true and when it is false during proof process
- Much harder for loops!

Axiomatic semantics: summary

- Given formal specification of program P:
  - ⇒ should be possible to prove P is correct
- However: very difficult, tedious in practice
  - Hard to develop axioms/inference rules for all statements in a language
  - Proof in predicate calculus is *exponential, semi-decidable*
- Good for reasoning about programs
- Not too useful for users or compiler writers
- Tools supporting axiomatic semantics: Java Modeling Language (JML), Haskell, Spark

Semantics

- Given M_s, the denotational semantics mapping function for a statement, come up with M_{sl}, the mapping function for a *list* of statements
- Find an axiomatic precondition for the following, if the postcondition $Q = \{y = 15\}$:

```
for (i=0,i<3,i++)
   y = y + x;
```

Is there only one?

Semantics

- Each group: assigned *operational, denotational, or axiomatic* semantics
- You will defend your assignment as the best approach to axiomatic semantics
- Make a brief statement; then other groups will attack/argue (you'll have a chance to return the favor)