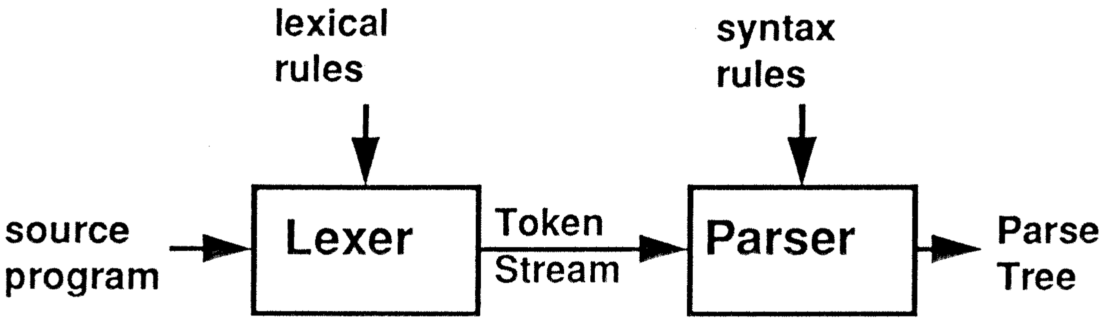
Syntax and Parsing

- *Syntax*: the form of a program
- *Semantics*: the meaning of a program
- Two parts to syntax analysis:
  - lexical rules: define legal characters and how they ca[n] be combined to form symbols

    ("lexemes").
  - syntax rules: define how categories of lexemes ("tokens") can be combined to form legal programs.

Compiler Front End



Syntax Analysis

- **We won't make a strict distinction, but will generally deal with syntax rules.**

How to Describe the Syntax of a Language?

- **English description**
  - lengthy, tedious, ambiguous
- **Formal description**
  - recognizer: given a string, a recognizer for a langu
    tells whether or not the string is in L
  - generator: a generator for L will produce an arbitra
    string in L on demand.
- **Recognition and generation are useful for differe things, but are closely related.**
- **First, we'll talk about an important generation to BNF.**

BNF

- Backus-Naur Form (BNF) is a *metalanguage* for describing the syntax of programming languages.
  - developed by John Backus and Peter Naur
  - first used to describe ALGOL60
- A language description in BNF is called a *grammar*.

Grammars

- **A grammar is made up of productions, or rules,**

  <sentence> --> <subject><verb><obj>

  <verb> --> see | hit

  <subject> --> I

  <object> --> him | her

- **Four components:**
  - **--> : "is defined as"**
  - **| : "or"**
  - **terminals : see, hit, I, him, her.**
  - **non-terminals : <sentence>, <verb>, <subject>,**

Recursion

- **Need recursion to define strings of indefinite length:**

<ones> --> 1 | 1<ones>

==> 1, 11, 111, 1111, . . .

<ablist> --> ab | a<ablist>b

==> ab

OR a<ablist>b

==> aabb

OR aa<ablist>bb

==> aaabbb

OR aaa<ablist>bbb

. . .

Derivations

- **The steps in generating a string from a grammar called a *derivation*.**

<exp> --> <id> | <exp> + <exp> | <exp> * <exp> | (<exp>)

<id> --> A | B | C

<exp> ==> <exp> + <exp>

==> <exp> * <exp> + <exp>

==> <id> * <exp> + <exp>

==> A * <exp> + <exp>

==> A * <id> + <exp>

==> A * B + <exp>

==> A * B + <id>

==> A * B + C

- **Each step produces a *sentential form*.**

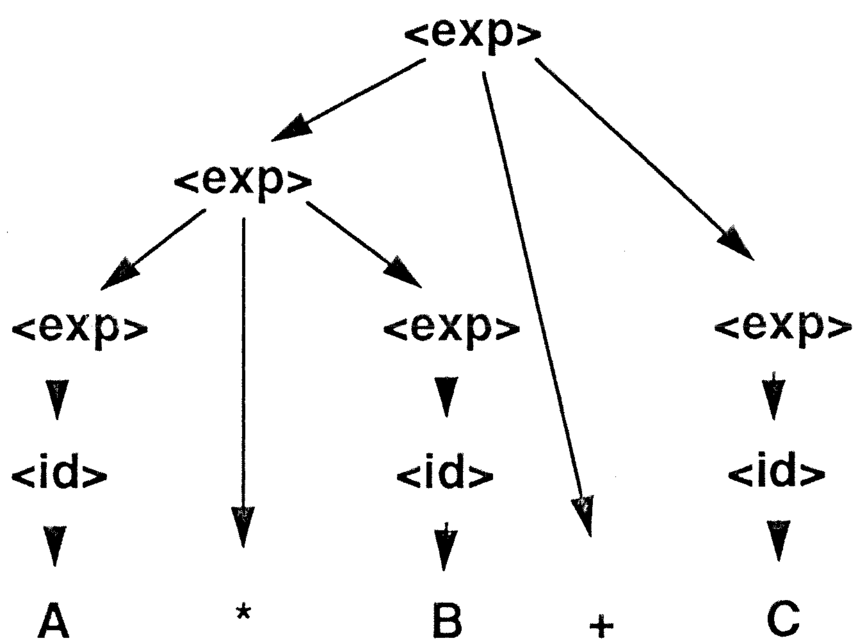
Left-Most and Right-Most Derivations

- **How to choose which non-terminal to replace next:**
  - **left-most derivation: replace left-most NT first**
  - **right-most derivation: replace right-most NT first**
- **Need not be either of these**
  - **random replacement OK**

    can't affect language generated

Parse Trees

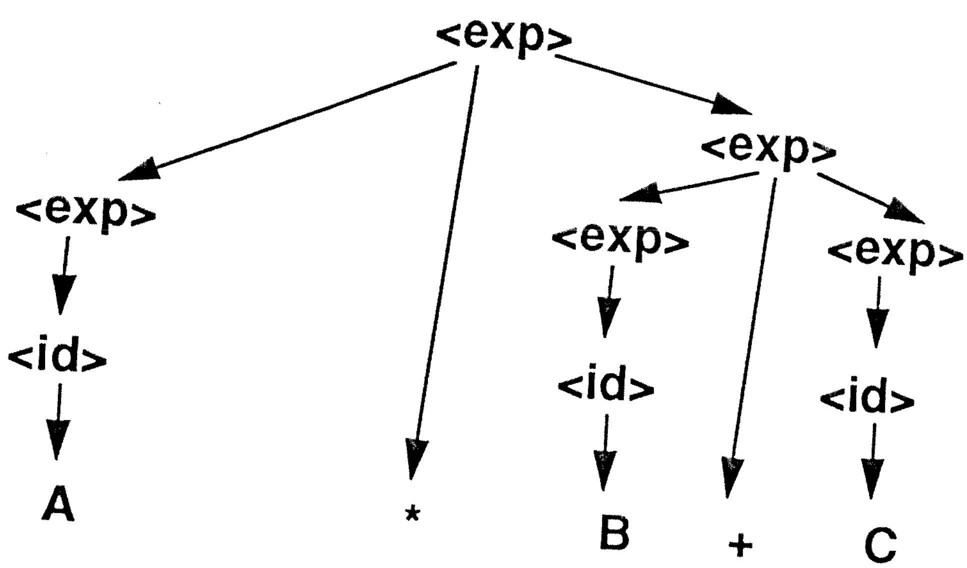
- **Show the <u>syntactic</u> structure of sentences.**



Ambiguous Grammars

- A grammar is ambiguous if it generates a sentence for which there are two or more parse trees.

- Another parse tree for A * B + C:



Disambiguating the Grammar

- **To disambiguate this grammar, change to:**

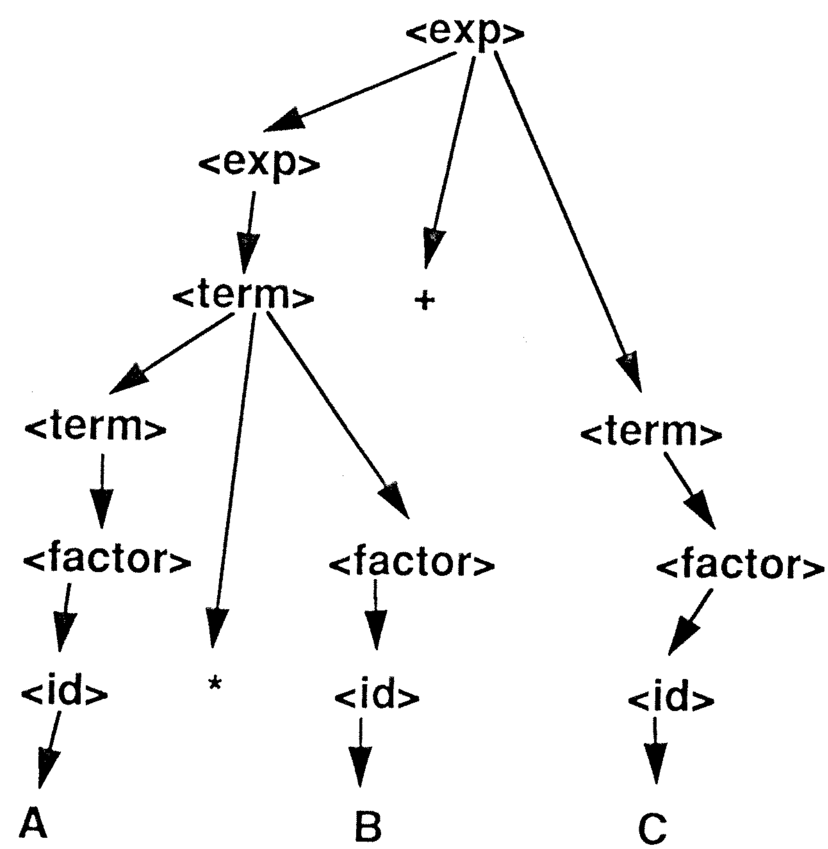
  <exp> --> <exp> + <term> | <term>

  <term> --> <term> * <factor> | <factor>

  <factor> --> (<exp>) | <id>

  <id> --> A | B | C

- **This gives * higher precedence than +, although i generates the same language as the first gramm**



Another Ambiguous Grammar

<stmt> --> <assign> | <if_stmt>

<assign> --> <id> := <exp>

<if_stmt> --> IF <bool> THEN <stmt> |

IF <bool> THEN <stmt> ELSE <stmt>

- **Exercise:**
  - **Prove that this is ambiguous.**
  - **Write a grammar for the <u>same</u> language that is not ambiguous.**

Limitations of Context Free Grammars

- **Productions must always apply, regardless of cont in which string appears.**

- **Can't handle some things:**

```
var x : integer;
    y : boolean;
begin
    x := 3;          OK
    x := y;          types wrong
    z := 5;          var z undeclared
```

- **Need "static semantics" . . .**

Recognizers

- How to generate a recognizer from a grammar?
  - automatically (YACC)
  - by hand
- There are many types of parsers:
  - LL(0)
  - LL(k)
  - LR
  - LALR
  - recursive descent

Extended BNF

- **[. . .] optional**

  <if> --> IF <bool> THEN <stmt> [ELSE <stmt>]

- **{. . .} zero or more times**

  <ones> --> 1 {<ones>}

- **(. . . | . . .) or (: : :) local choice**

  <exp> --> <id> | <exp> (+ | *) <exp>

  or $\left(\begin{array}{c} + \\ * \end{array}\right)$

Syntax and Parsing Summary

- **Recognizer vs. generator**
- **BNF**
  - **Four components**
  - **Recursion**
- **Derivation**
- **Ambiguous grammars**
- **Extended BNF**

Semantics of Programming Languages

- **How to define the meaning of programs?**
- **Three approaches:**
  - Operational
  - Axiomatic
  - Denotational

Operational Semantics

- Gives a program's meaning in terms of its implementation on a real or virtual machine
- Define two parts:
- machine
  - high level
  - low level
- translation from source code to "machine" code

Example

Pascal	Operational Semantic

for i := x to y do
begin
.
.
.
end
	i := x
	loop: if i>y goto out
	.
	.
	.
	i := i + 1
	goto loop
	out: . . .

- **Operational semantics could be much lower le e.g.,**

```
      mov i,r1
      mov y,r2
      jmpifless(r2,r1,out)
      ...
out:  ...
```

Advantages and Disadvantages of Operational Semantics

- **Advantages:**
  - **May be simple, intuitive for small examples**
  - **Useful for implementation**
- **Disadvantages**
  - **Very complex for large programs**
  - **Lacks mathematical rigor**
- **Uses:**
  - **Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)**
  - **Compiler work**

Axiomatic Semantics

- **Based on predicate calculus. Use assertions to s certain properties of programs.**

  {P} statement {Q}

- **Compute precondition from postcondition:**

  {P} x := y + 1 {X>0}

  - **Possible Ps:**

    y > 5

    y = 37 **Weakest Precondition (WP)**

    y ≥ 0

    etc.

- ***WP* -> identifies *all possible* cases for which postcondition holds!**

Finding the Weakest Precondition

- **Define function:**

  wp: Stmt x Postcondition --> weakest precondition

  stmt   post condition

  wp (x := e, P) = $P_{x \to e}$ "substitute e for every x in P"

- **So:**

  wp (x := y+1, x > 0)

  $= x > 0_{\,x \to y+1}$

  $= y+1 > 0$

  $= y \geq 0$

  - **basically, "undoing" the assignment and solving for y**

Sequences of Statements

{P} S1; S2 {Q}

- **Just apply wp twice**

$$wp\ (x := y + 1;\ z := x + y,\ z > 5)$$

$$wp\ (z := x + y,\ P1) \qquad \text{where}$$

$$P1 = z > 5_{\ x \to y+1}$$

$$= x + y > 5$$

$$wp\ (x := y + 1,\ x + y > 5)$$

$$= x + y > 5_{\ x \to y+1}$$

$$= y + 1 + y > 5$$

$$= y > 2$$

Loops

- {P} while B do S end {Q}
- Need *loop invariant* I such that:
  - P ==> I
  - {I} B {I}
  - {I & B} S {I}
  - (I & (not B)) ==> Q
  - and the loop terminates

Finding Loop Invariants

- **Work backwards through a few iterations and l a pattern.**

while y <> x do y:= y+1 {y = x}

wp (y := y + 1, {y = x}) = {y = x} $_{y \to y+1}$

= y = x - 1 -- last time

wp (y := y + 1, {y = x - 1}) = {y = x-1} $_{y \to y+1}$

= y = x - 2 -- next to l

I = {y ≤ x } -- by exten

- **This also satisfies loop termination, so**

P = I = {y ≤ x}

- **It's not always this easy!**

Finding Loop Invariants (cont.)

{P} while y < x + 1 do y := y + 1 {y>5}

$y>5_{\ y \rightarrow y+1} \quad \Rightarrow y>4$

$y>4_{\ y \rightarrow y+1} \quad \Rightarrow y>3$

etc.

- **really tells us *nothing* relative to x because x is not in Q ≡ {y > 5}**

- **Try Using Boolean**

  I & (not B) => Q

  ? & y ≥ x + 1 => y > 5

  ? & y > x => y > 5

  any x ≥ 5 satisfies implication

  so . . . let $\boxed{I \equiv X \geq 5}$

- **Do the 4 Axioms hold?**

Advantages, Disadvantages, and Uses o Axiomatic Semantics

- **Advantages**
  - **Can be very abstract**
  - **May be useful in proofs of correctness**
  - **Solid theoretical foundations**
- **Disadvantages**
  - **Predicate transformers are hard to define**
  - **Hard to give complete meaning**
  - **Does not suggest implementation**
- **Uses of Axiomatic Semantics**
  - **Semantics of Pascal**
  - **Reasoning about correctness**

HOMEWORK FOR AXIOMATIC SEMANTICS

Consider

$\{P\}$ x := x * 3 $\{X^2 = 36\}$

Determine <u>Weakest Precondition</u> for $\{P\}$

Denotational Semantics

- Define a function that maps a program (a synt
object) to its meaning (a semantic object).

- Sort of like a high-level operational semantics
  - → machine is gone
  - → language is λ-calculus

- More abstract.

Example: Decimal Numbers

Valuation function: V: Number --> Integers

syntax meaning

- **Syntax:**

  <num> --> <num> <digit> | <digit>

  <digit> --> 0 | 1 | 2 | . . . | 9

- **Semantics:**

  → **Let n ∈ <num>, d ∈ <digit>**

  $V[\![nd]\!] = 10 * V[\![n]\!] + V[\![d]\!]$

  $V[\![0]\!] = 0$

  integers as we know them

  $V[\![1]\!] = 1$

- **Consider $V[\![237]\!]$:**

  $V[\![237]\!] = 10 * V[\![23]\!] + V[\![7]\!]$

  $= 10 * (10 * V[\![2]\!] + V[\![3]\!]) + V[\![7]\!]$

  $= 10 * (10 * 2 + 3) + 7$

  $= 10 * (20 + 3) + 7$

  $= 10 * (23) + 7$

  $= 230 + 7$

  $= 237$

Expressions

- **But for *real* programming languages we need m info:**

  E: Expression --> Integer

  $E[\![x]\!] = ?$ where x is a variable

- **Depends on the current *state***

  → **STATE = <mem, input, output>**

  mem: Identifier --> Integer

  input: Integer *

  output: Integer *

- **Now**

  E: Expression x STATE --> Integer

  $E([\![x]\!], s) = mem([\![x]\!])$ where s = <mem,i,o>

  $E([\![e_1 + e_2]\!], s) = E([\![e_1]\!], s) + E([\![e_2]\!], s)$

Statements

- **Expressions denote a value, but statements denote a state.**

ST: (Stmt x STATE) --> STATE

$ST(\llbracket x := e \rrbracket, s) = \langle mem', i, o \rangle$ where

$s = \langle mem, i, o \rangle$

$mem' \llbracket x \rrbracket = E(\llbracket e \rrbracket, s)$

$mem' \llbracket y \rrbracket = mem \llbracket y \rrbracket$ for all $y \neq x$

$ST(\llbracket write(e) \rrbracket, s) = \langle mem, i, o' \rangle$ where

$s = \langle mem, i, o \rangle$

$o' = o \bullet (E(\llbracket e \rrbracket, s))$

Sequences of Statements

- **Basic (sequential statement evaluation)**

  $ST(\llbracket stmt_1;\ stmt_2 \rrbracket, s) =$

  $ST(\llbracket stmt_2 \rrbracket, s')$ where

  $s' = ST(\llbracket stmt_1 \rrbracket, s)$

- **Parallel statement evaluation**

  $ST(\llbracket stmt_1;\ stmt_2 \mid\mid stmt_3 \rrbracket, s) = \{s_1, s_2, s_3\}$ where

  $s_1 = ST(\llbracket stmt_1; stmt_2; stmt_3 \rrbracket, s)$

  $s_2 = ST(\llbracket stmt_1; stmt_3; stmt_2 \rrbracket, s)$

  $s_3 = ST(\llbracket stmt_3; stmt_1; stmt_2 \rrbracket, s)$

Example

```
P: {           x := 5;
      P': {    y := x + 1;
               write(x * y);   }  P''
```

→ **Initial state s = <mem,i,o>**

$$ST(\llbracket P \rrbracket\ , s) = ST(\ \llbracket P' \rrbracket\ , \underbrace{(ST(\llbracket x := 5 \rrbracket\ , s))}_{s'})$$

$s' = \langle mem', i', o' \rangle$ where

$$mem'(\llbracket x \rrbracket) = 5$$

$$mem'(\llbracket z \rrbracket) = mem(\llbracket z \rrbracket) \qquad \text{for all } z \neq x$$

$$i' = i,\ o' = o$$

Example (continued)

- $\mathbf{ST(\llbracket P' \rrbracket, s') = ST(\llbracket P'' \rrbracket, \underbrace{(ST(\llbracket y := x + 1 \rrbracket, s'))}_{s''})}$

  $s'' = \langle mem'', i'', o'' \rangle$ where

  $mem''(\llbracket y \rrbracket) = E(\llbracket x + 1 \rrbracket, s') = 6$

  $mem''(\llbracket z \rrbracket) = mem'(\llbracket z \rrbracket)$ for all $z \neq y$

  $i'' = i', o'' = o'$

- $\mathbf{ST(\llbracket P'' \rrbracket, s'') = ST(\llbracket write\ (x * y) \rrbracket, s'') = s'''}$

  $s''' = \langle mem''', i''', o''' \rangle$ where

  $mem''' = mem'', i''' = i''$

  $o''' = o'' \bullet E(\llbracket x * y \rrbracket, s'') = o'' \bullet 30$

- **So,**

  $ST(\llbracket P \rrbracket, s) = \langle mem''', i''', o''' \rangle$ where

  $mem'''(\llbracket y \rrbracket) = 6$

  $mem'''(\llbracket x \rrbracket) = 5$

  $mem'''(\llbracket z \rrbracket) = mem(\llbracket z \rrbracket)$ for all $z \neq x,y$

  $i''' = i$

  $o''' = o \bullet 30$

Advantages, Disadvantages, and Uses of Denotational Semantics

- Advantages (of denotational semantics)
  - → compact and precise
  - → may help with implementation
  - → solid mathematical foundations
- Disadvantages
  - → Hard for programmer to use
- Uses
  - → Semantics for Algol-60, Pascal, etc.
  - → Compiler generation and optimization

HOMEWORK FOR DENOTATIONAL SEMANTICS

Prefatory Consideration:

Prog. Langs. have conditional statements, e.g.

1) if b then stmt1, else stmt2

2) if b then exp1, else exp2

Assuming that conditionals only support expres
evaluation and have no side effects, let's give meaning to 1) above:

ST(⟦ if b then stmt, else stmt2⟧ , s) =

if E(⟦b⟧ , s) then

ST(⟦stmt1⟧ , s) else

ST(⟦stmt2⟧ , s).

Note: use of previous defns T/F Assessment like introduction of"IF THEN ELSE" in denotational language

UNDERSTAND/STUDY

1) ST (⟦ if b then stmt1 else stmt2⟧ , s)

definition and elaboration

2) Give denotational semantics for repeat until stmt

<u>REPEAT stmt UNTIL b</u>

You will need conditional statement like that specified above

HINT:

on RHS you ight find <u>recursive defn</u>