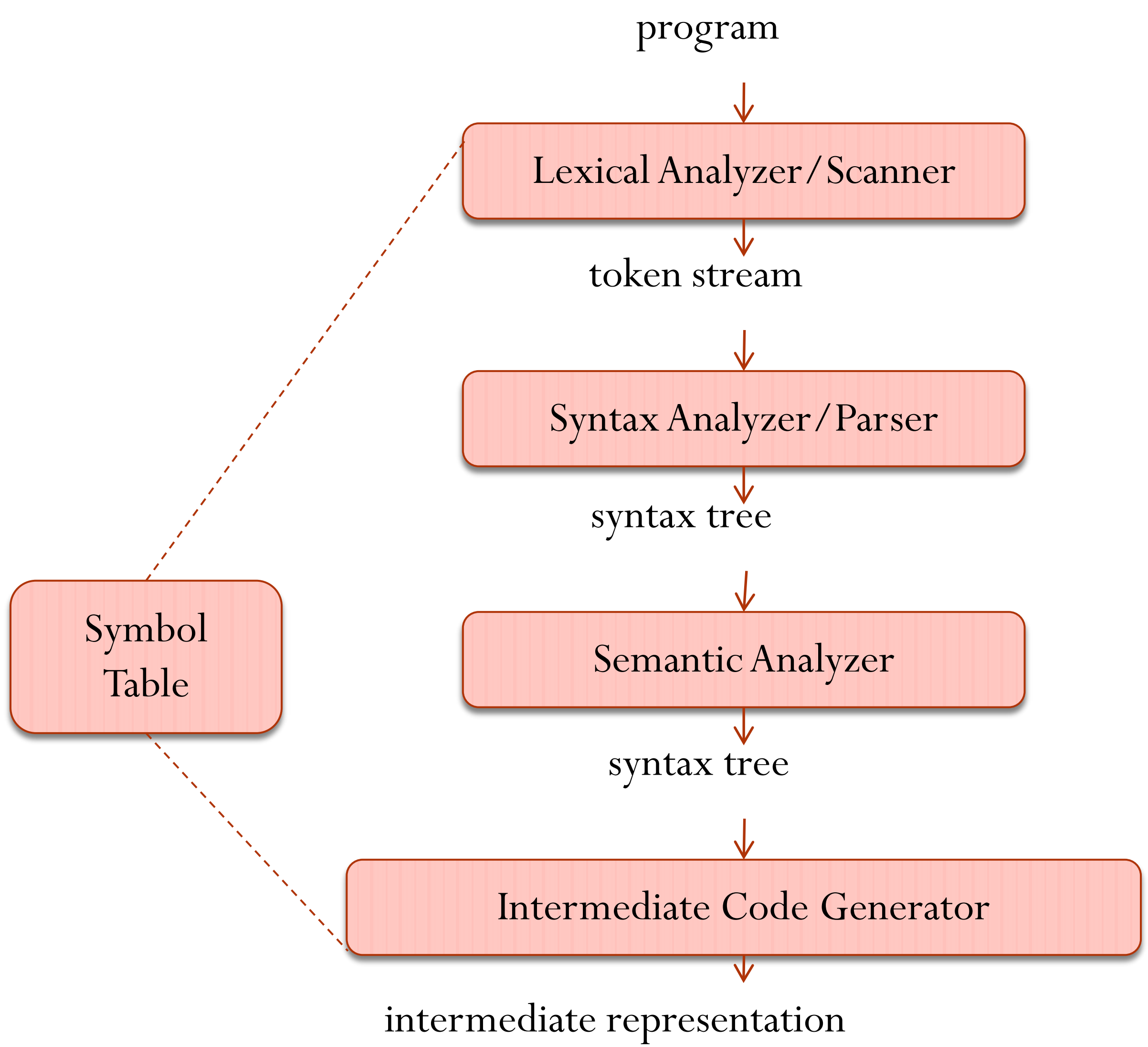
Compilers

Topic: Parsing

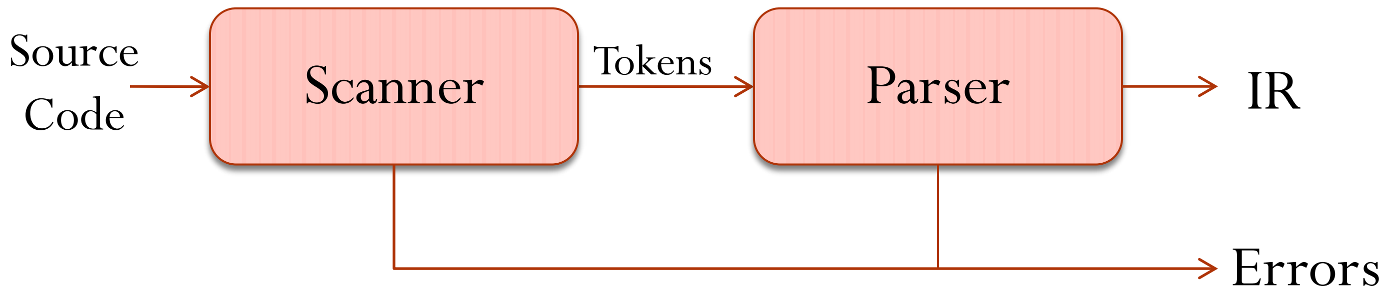
Monsoon 2011, IIIT-H, Suresh Purini

ACK: Some slides are based on Keith Cooper's CS412 at Rice University

The Front End



The Front End: Scanner and Parser



Parser

- Takes as input a stream of tokens
- Checks if the stream of tokens constitutes a syntactically valid program of the language
- If the input program is syntactically correct
  - Output an intermediate representation of the code (like AST)
- If the input program has syntactic errors
  - Outputs relevant diagnostic information

Context Free Grammars and Programming Languages

Expr → Expr Binop Expr | − Expr | ! Expr | (Expr)

Binop → Arithop | Relop | Eqop | Condop

Arithop → + | − | * | / | % | << | >>

Relop → < | > | <= | >=

Eqop → == | !=

Condop → && | ||

CFGs and Programming Languages

$$
\begin{aligned}
\text{Statement} \rightarrow\ & \text{Location} = \text{Expr} \ ; \\
| \ & \text{MethodCall} \ ; \\
| \ & \text{if} \ (\ \text{Expr} \ ) \ \text{Block} \\
| \ & \text{if} \ (\ \text{Expr} \ ) \ \text{Block} \ \ \text{else} \ \ \text{Block} \\
| \ & \text{while} \ (\ \text{Expr} \ ) \ \ \text{Block} \\
| \ & \text{continue} \ ; \\
| \ & \text{Block}
\end{aligned}
$$

$$\text{Block} \rightarrow \{ \ \ \text{VarDeclList} \ \ \text{StatementList} \ \ \}$$

$$\text{StatementList} \rightarrow \text{Statement} \ \ | \ \ \text{Statement} \ \ \text{StatementList}$$

Context Free Grammars and Programming Languages

Key Idea: All modern programming languages can be expressed using context free grammars (by design!)

- Programs have recursive structures
  - A program is a collection of functions
  - A function is a sequence of statements
  - A statement can be any of if, while, for, assignment statements etc.
  - The body of a while loop is a sequence of statements
  - An arithmetic expression is a sum/product of two AEs.

CFGs are a nice way of expressing programs with recursive structure

CFGs and Programming Languages

Advantages of using CFGs to specify syntactic structure of languages

- Clear and concise syntactic specification for languages
- Language can be developed or evolved iteratively
  - New constructs in the language can be added with relative ease.
- Programming languages can be specified using a special sub-class of CFGs for which efficient parsing techniques and automatic parser generators exists.
  - These special class of CFGs also allow for automatically capturing ambiguities in the language
- CFGs impose a structure on the program which facilitates easy translation to intermediate or target object code.

Syntax Specification Using Context Free Grammars

1. Goal $\longrightarrow$ Expr
2. Expr $\longrightarrow$ Expr Op Term | Term
3. Term $\longrightarrow$ number | id
4. Op $\longrightarrow$ + | -

S = Goal (Start Symbol)

T = { number, id, +, - }

N = { Goal, Expr, Term }

Question: Given a stream of tokens (read terminals) and the syntax specification in the form a CFG, how can the parser check the syntactic correctness of the source code?

Formally, a grammar G = (S,N,T,P)

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of productions or rewrite rules
  - Each production is of the form $A \rightarrow \alpha$ where $A \in N$ and $\alpha \in (N \cup T)^*$.

Derivations and Parsing

Def: The process of deriving strings by applying productions in the grammar is called derivation.

Def: The Process of discovering the derivation is called Parsing

Example:

Goal ⇒ Expr ⇒ Expr Op Term ⇒ Term Op Term ⇒ id Op Term ⇒ id + Term ⇒ id + number

At each step of Derivation two questions to answer

1. Which Non-Terminal Symbol to replace?
2. Which Substitution Rule to apply for the chosen non-terminal symbol?

Derivation

Question 1: Which Non-Terminal symbol to replace?

- Left-most derivation – We replace the left-most non-terminal at every step of derivation. (Used in Top-Down Parsing Approach)
- Right-most derivation – We replace the right-most non-terminal at every step of derivation (Used in Bottom-Up Parsing Approach)

We don't care about the Randomly-Ordered Derivations

Derivations and Sentential Forms

Given a derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots\ldots\ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

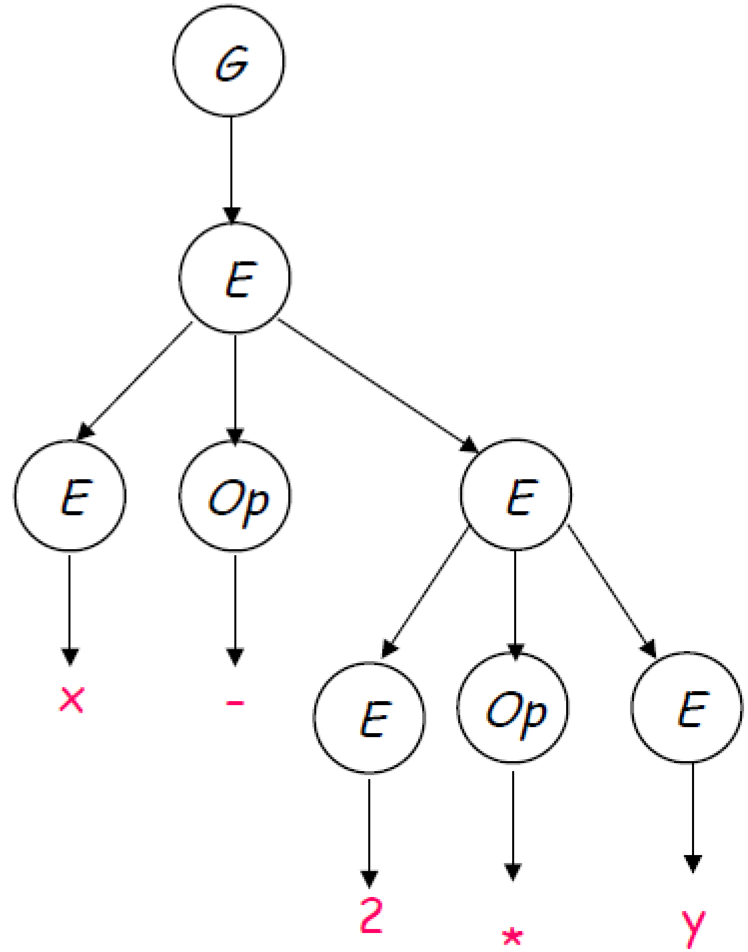
- $\gamma_0, \gamma_1, \ldots, \gamma_n \in (N \cup T)^*$ are called sentential forms.
- Notation: $S \Rightarrow^* w$
- If the derivation is a left-most derivation, then $\gamma_0, \gamma_1, \ldots, \gamma_n$ are called left-sentential forms.
- If the derivation is a right-most derivation, then $\gamma_0, \gamma_1, \ldots, \gamma_n$ are called right-sentential forms.

Derivations and Parse Trees

Left-most Derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	<id,x> *Op Expr*
5	<id,x> - *Expr*
1	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/



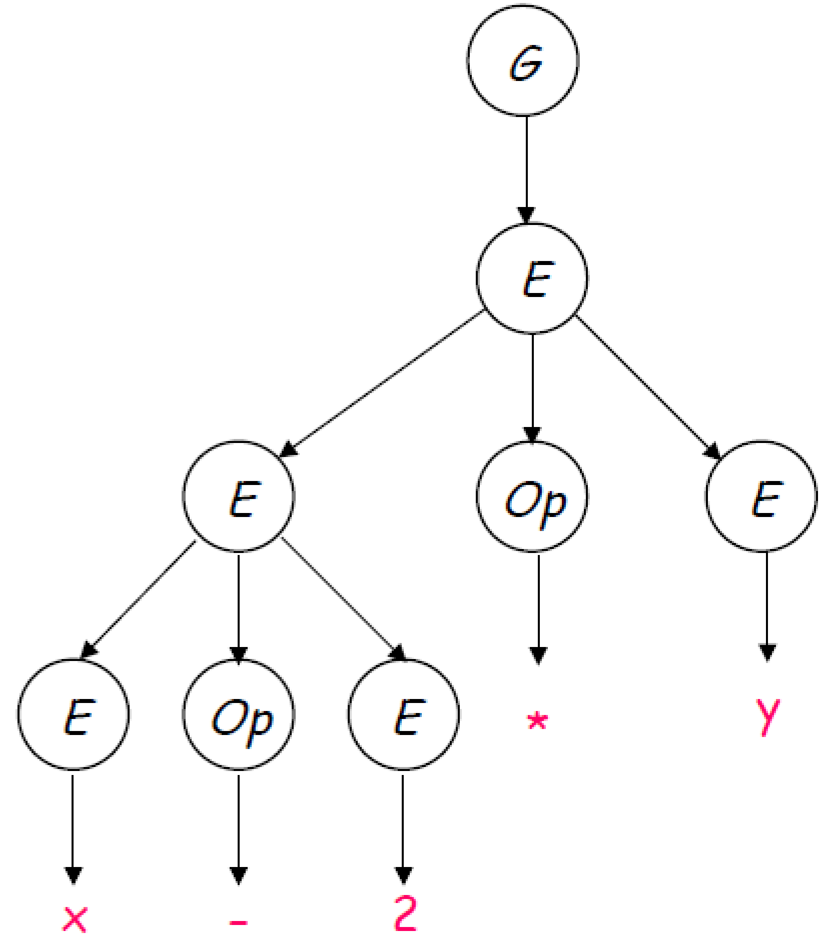
This evaluates as x - (2 * y)

Derivations and Parse Trees

Right-most Derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* <id,y>
6	*Expr* * <id,y>
1	*Expr Op Expr* * <id,y>
2	*Expr Op* <num,2> * <id,y>
5	*Expr* - <num,2> * <id,y>
3	<id,x> - <num,2> * <id,y>

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/



This evaluates as (x - 2) * y

Parse Trees and Ambiguity in Semantics

Key Idea: Parse trees not only capture the syntax of a program but also encode its semantics.

Def: A grammar is ambiguous if and only if there exists more than one parse trees for a sentence.

Derivations and Parse Trees

- Multiple derivations can lead to the same parse tree.
  - Many-to-one mapping between derivations and parse trees
- Every left-most derivation has a corresponding unique parse tree
  - one-to-one mapping between left-most derivations and parse trees
- Every right-most derivation has a corresponding unique parse tree
  - one-to-one mapping between right-most derivations and parse trees
- Def: A grammar is ambiguous if and only if there exists more than one left-most (or right-most) derivation for a sentence.

Parse Trees and Ambiguity

Examples:

Grammar 1

1. Expr $\rightarrow$ Expr + Expr
2. Expr $\rightarrow$ Expr * Expr
3. Expr $\rightarrow$ id

Grammar 2

1. Stmt $\rightarrow$ if Expr then Stmt else Stmt
2. Stmt $\rightarrow$ Stmt if Expr then Stmt
3. Stmt $\rightarrow$ otherStmt

- Grammar 1 doesn't capture the Operator Precedence Rules and Associativity Rules.
- What's the problem with the Grammar 2?

Ambiguity in the Expression Grammar

Question: How to add precedence and associativity rules?

1. $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E)$
2. $E \rightarrow id \mid num$

To add precedence

- Create a non-terminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions

- Multiplication and division, first (level one)
- Subtraction and addition, next (level two)

Derivations and Precedence

- Adding the standard algebraic precedence produces:

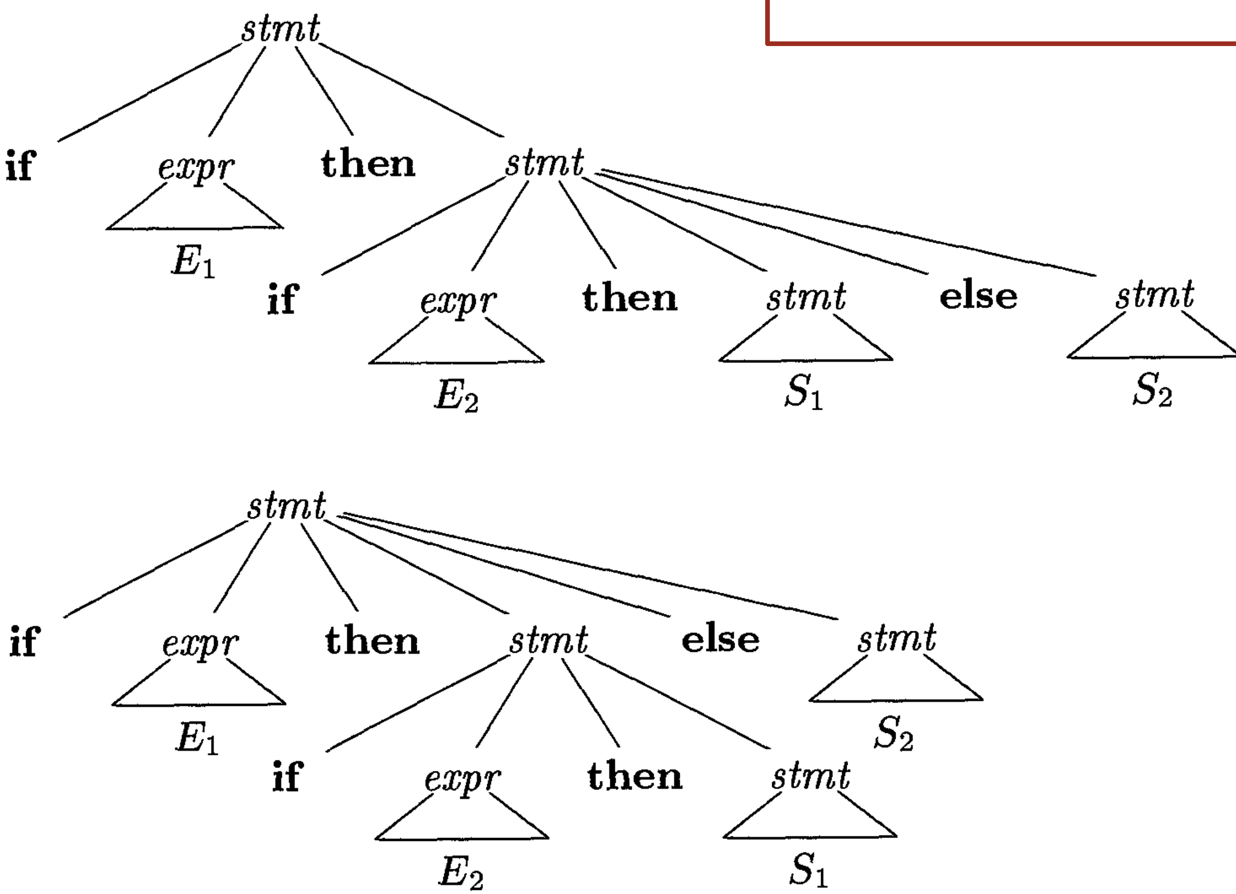
1.	$E \rightarrow E + T \mid E - T \mid T$	(Level 2)
2.	$T \rightarrow T * F \mid T / F \mid F$	(Level 1)
3.	$F \rightarrow (E) \mid id \mid num$	(Level 0)

- Grammar is slightly larger
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence and associativity rules

Ambiguity in if-then-else Statements

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

$\boxed{\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2}$



if-then-else Statement – Unambiguous Grammar

- The following grammar associates the else part with the closest unmatched if.

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ matched\_stmt\ \textbf{else}\ matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if}\ expr\ \textbf{then}\ stmt \\
 & | & \textbf{if}\ expr\ \textbf{then}\ matched\_stmt\ \textbf{else}\ open\_stmt
\end{array}
$$

Deeper Ambiguity

- Productions involving Procedure Calls and Array References

$$
\begin{array}{lrcl}
(1) & stmt & \rightarrow & \textbf{id} \; (\; parameter\_list \;) \\
(2) & stmt & \rightarrow & expr \; := \; expr \\
(3) & parameter\_list & \rightarrow & parameter\_list \; , \; parameter \\
(4) & parameter\_list & \rightarrow & parameter \\
(5) & parameter & \rightarrow & \textbf{id} \\
(6) & expr & \rightarrow & \textbf{id} \; (\; expr\_list \;) \\
(7) & expr & \rightarrow & \textbf{id} \\
(8) & expr\_list & \rightarrow & expr\_list \; , \; expr \\
(9) & expr\_list & \rightarrow & expr
\end{array}
$$

Ambiguity – Summary

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  - Knowledge of declarations, types, …
  - Accept a superset of L(G) & check it by other means†
  - This is a language design problem
- Sometimes, the compiler writer accepts an ambiguous grammar
  - Parsing techniques that "do the right thing"
  - i.e., always select the same derivation

Non-Context-Free Language Constructs

- Can we capture the constraint that a variable has to be declared before it is used for the first time in a CFG?
- How to check that the number and the type of parameters match between a function call and a function declaration?