Introduction to Parsing

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFG’s)
- Derivations
- Ambiguity

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages, tree languages

Beyond Regular Languages

- Many languages are not regular
- Strings of balanced parentheses are not regular:

$$\{(^i)^i \mid i>=0\}$$

- There are many similar constructs in programming languages that cannot be handled with regular expressions
- E.g., nested if statements

What Can Regular Languages Express?

- So what can regular languages express?
- Consider the following FA

What can it do?

- It can tell if the number of ones in the input is divisible by 2
- i.e. it can count mod 2
- In general a FA can count mod k, where k is the number of states
- but cannot remember how many ones it has seen
- Therefore it cannot express $(^i\ )^i$
- Languages requiring counting modulo a fixed integer
- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state

The Functionality of the Parser

- Input: sequence of tokens from lexer

- Output: parse tree of the program

(But some parsers never produce a parse tree . . .)

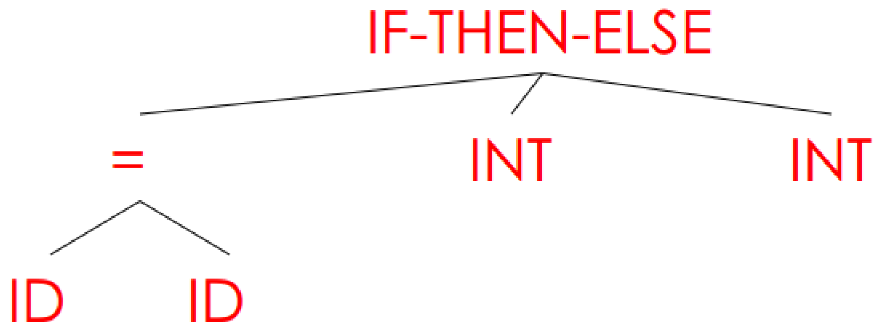
Example

if x = y then 1 else 2 fi

- Parser input

IF ID = ID THEN INT ELSE INT FI

- Parser output



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

The Role of the Parser

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens
- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

Context-Free Grammars

- Programming language constructs have recursive structure .

- An EXPR is

  if EXPR then EXPR else EXPR fi

  while EXPR loop EXPR pool

  …

- Context-free grammars are a natural notation for this recursive structure

CFGs (Cont.)

- A CFG consists of
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S(a non-terminal)
  - A set of productions

$$\mathbf{X \longrightarrow Y_1\ Y_2 \ldots Y_n}$$

$$\text{where } X \in N \text{ and } Y_i \in T \cup N \cup \{\varepsilon\}$$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

Terminals

- Terminals are so-called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

Examples of CFGs

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

Simple arithmetic expressions:

$$
\begin{aligned}
E \quad \rightarrow \quad & E * E \\
| \quad & E + E \\
| \quad & (E) \\
| \quad & id
\end{aligned}
$$

The Language of a CFG

- Read productions as rules (replacement rules):

$$X \rightarrow Y_1 \ldots Y_n$$

Means X can be replaced by $Y_1 \ldots Y_n$

Key Idea

1. Begin with a string consisting of the start symbol “S”
2. Replace any non-terminal X in the string by a the right-hand side of some production

   $$X \rightarrow Y_1 \ldots Y_n$$

3. Repeat (2) until there are no non-terminals in the string

The language of a CFG

- More Formally, write a single step

$$X_1 \ldots X_i \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

- If there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The language of a CFG

- Multiple steps (0 or more steps)

$\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$

$\alpha_0 \rightarrow^* \alpha_n$ (in 0 or more steps)

The Language of a CFG

- Let G be a context-free grammar with start symbol S. Then the language of G is:

$\{ \alpha_{1 \ldots} \alpha_n \mid S \rightarrow^* \alpha_{1 \ldots} \alpha_n$ and every α_i is a terminal$\}$

- What this says is the language of a CFG is the set of strings that can be derived starting from the start symbols and contain only terminal symbols.

Examples

$L(G)$ is the language of CFG G

Strings of balanced parentheses $\left\{ (^i)^i \mid i \geq 0 \right\}$

Two grammars:

$$S \rightarrow (S) \qquad\qquad S \rightarrow (S)$$

OR

$$S \rightarrow \varepsilon \qquad\qquad \mid \varepsilon$$

Examples of CFG

- Write a CFG that generates Even Palindrome

  $S \rightarrow aSa \mid bSb \mid \epsilon$

- Write a CFG that generates Odd Palindrome

  $S \rightarrow aSa \mid bSb \mid a \mid b$

- Write a CFG that generates Equal number of a’s and b’s

  $S \rightarrow aSbS \mid bSaS \mid \epsilon$

More CFG Examples

- Write a CFG that generates Equal number of a’s, b’s and c’s

  $S \rightarrow aSbScS \mid aScSbS \mid bSaScS \mid bScSaS \mid cSaSbS \mid cSbSaS \mid \epsilon$

Derivations and Parse Trees

- A derivation is a sequence of productions

  $S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$

- A derivation can be drawn as a tree
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

- Grammar

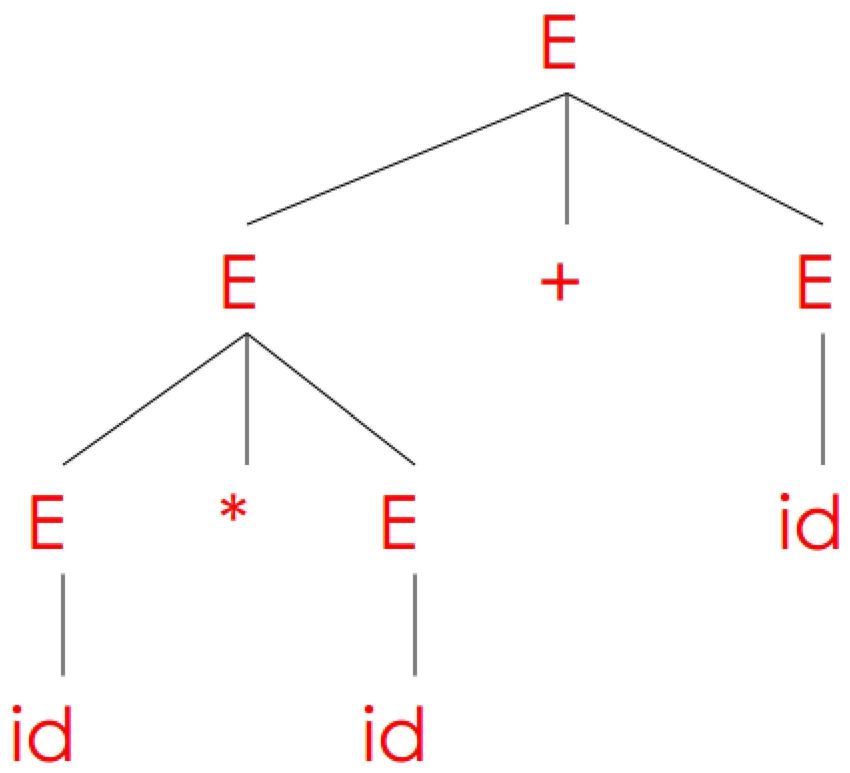
  E→ E +E | E*E| (E) | id

- String

  id * id + id

Derivation Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the precedence of operations, the input string does not

Left-most and Right-most Derivations

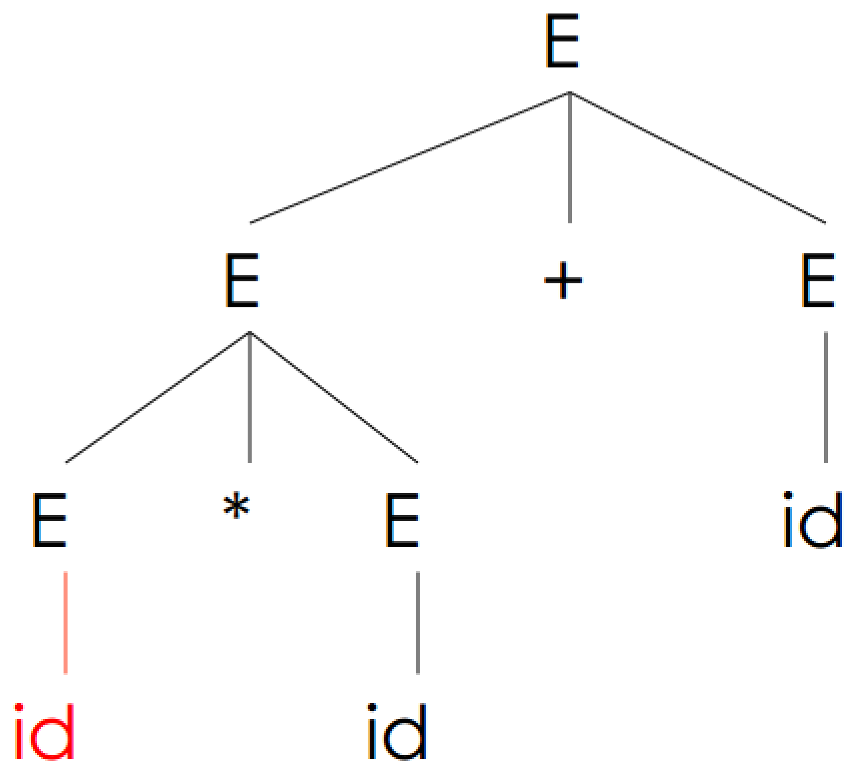
- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal

- There is an equivalent notion of a *right-most* derivation

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$

Right-most Derivation in Detail

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E * E + id \\
\rightarrow\ & E * id + id \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Summary of Derivations

- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar

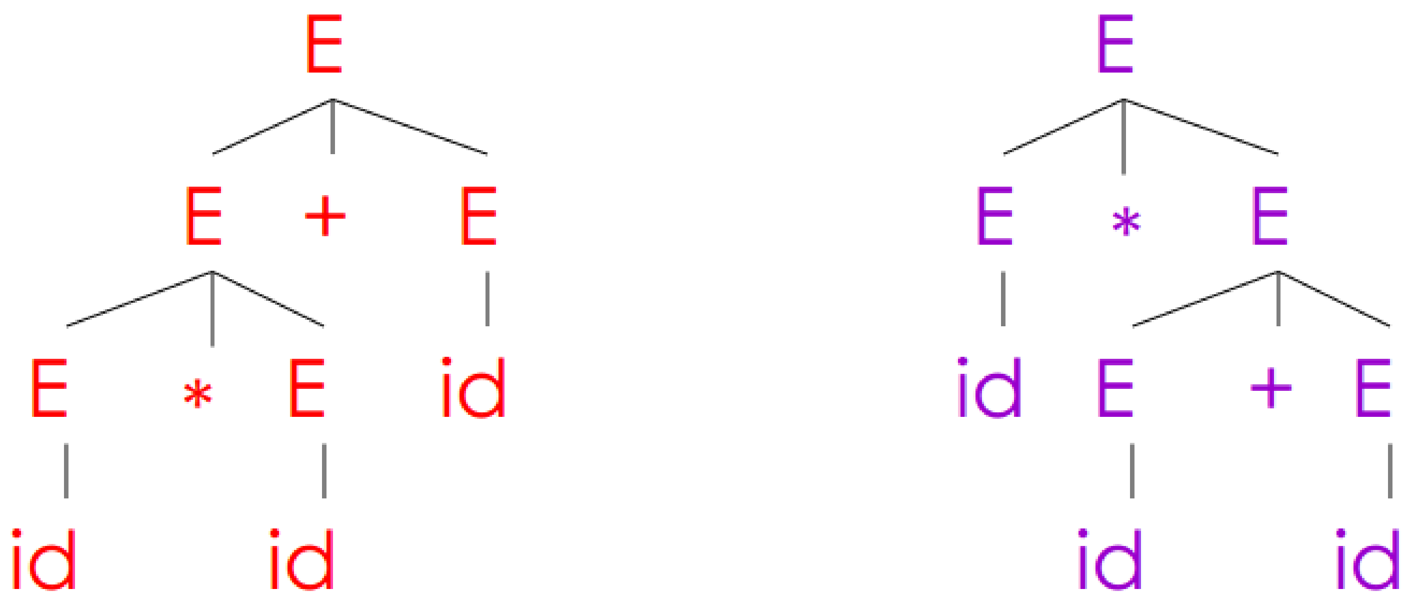
  $E \rightarrow E+E \mid E * E \mid (E) \mid id$

- String:

  id * id + id

Ambiguity

This string has two parse trees



Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string

- Ambiguity is BAD
  - Leaves meaning of some programs ill-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite grammar unambiguously

$$
\begin{aligned}
E &\rightarrow E' + E \mid E' \\
E' &\rightarrow id * E' \mid id \mid (E) * E' \mid (E)
\end{aligned}
$$

- Enforces precedence of * over +

Ambiguity: The Dangling Else

- Consider the grammar

$$E \rightarrow \text{if } E \text{ then } E$$
$$\mid \text{if } E \text{ then } E \text{ else } E$$
$$\mid \text{OTHER}$$

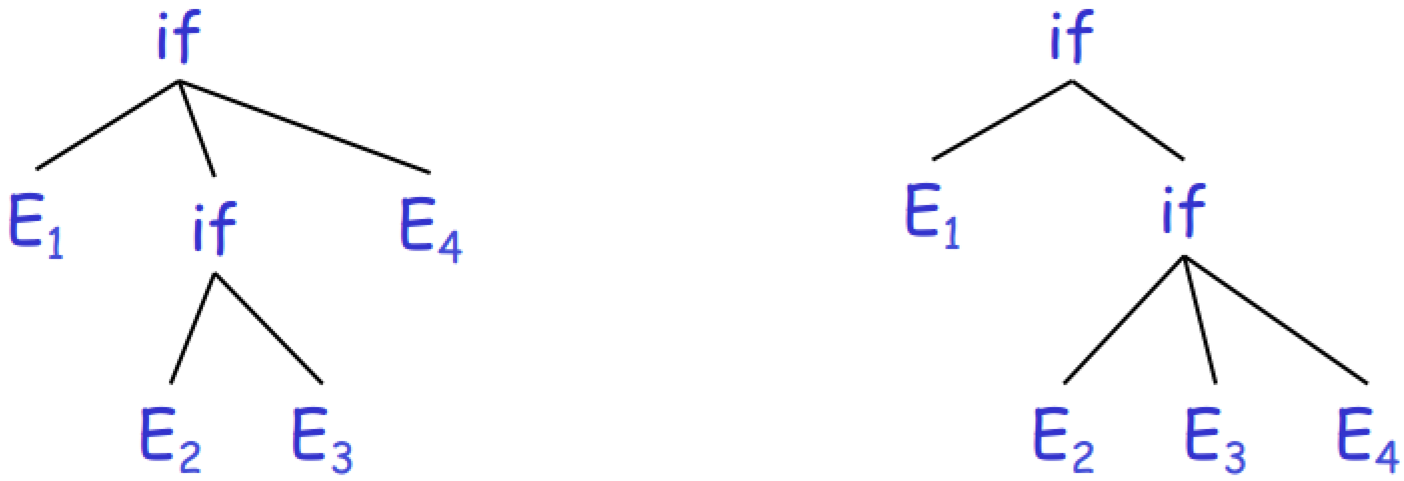
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  $\text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4$

  has two parse trees



- Typically we want the second form

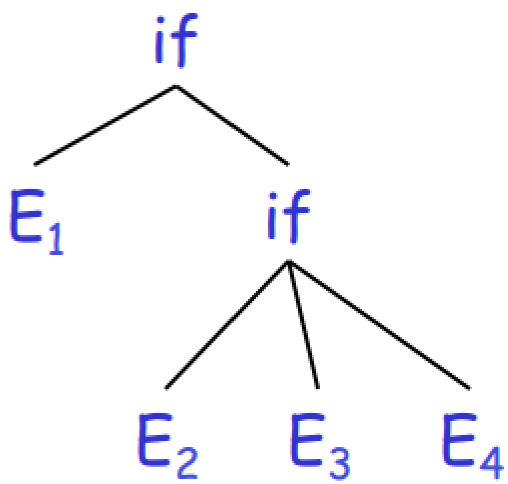
The Dangling Else: A Fix

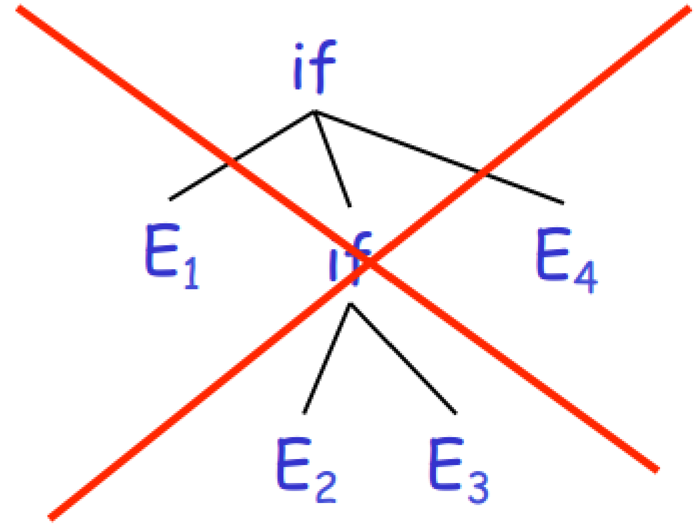
- else matches the closest unmatched then
- We can describe this in the grammar

```
E → MIF              /* all then are matched */
  | UIF              /* some then is unmatched */
MIF → if E then MIF else MIF
    | OTHER
UIF → if E then E
    | if E then MIF else UIF
```

- Describes the same set of strings

- The expression if E_1 then if E_2 then E_3 else E_4





- A valid parse tree (for a UIF)

- Not valid because the then expression is not a MIF

Ambiguity

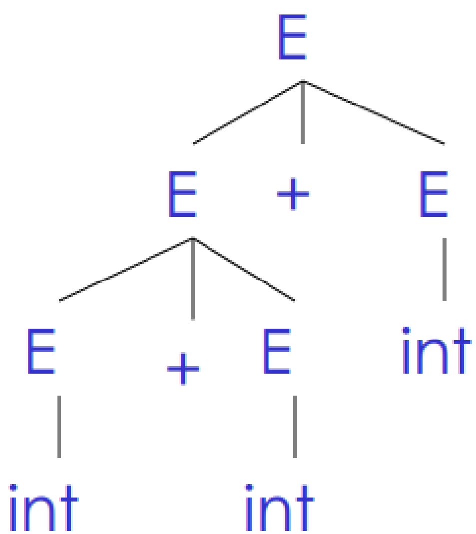
- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

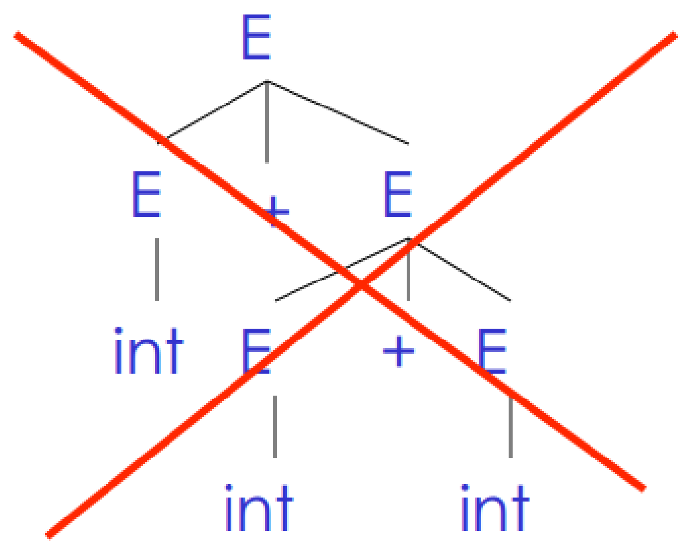
Precedence and Associativity Declarations

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars
- Examples …

Associativity Declarations

- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int

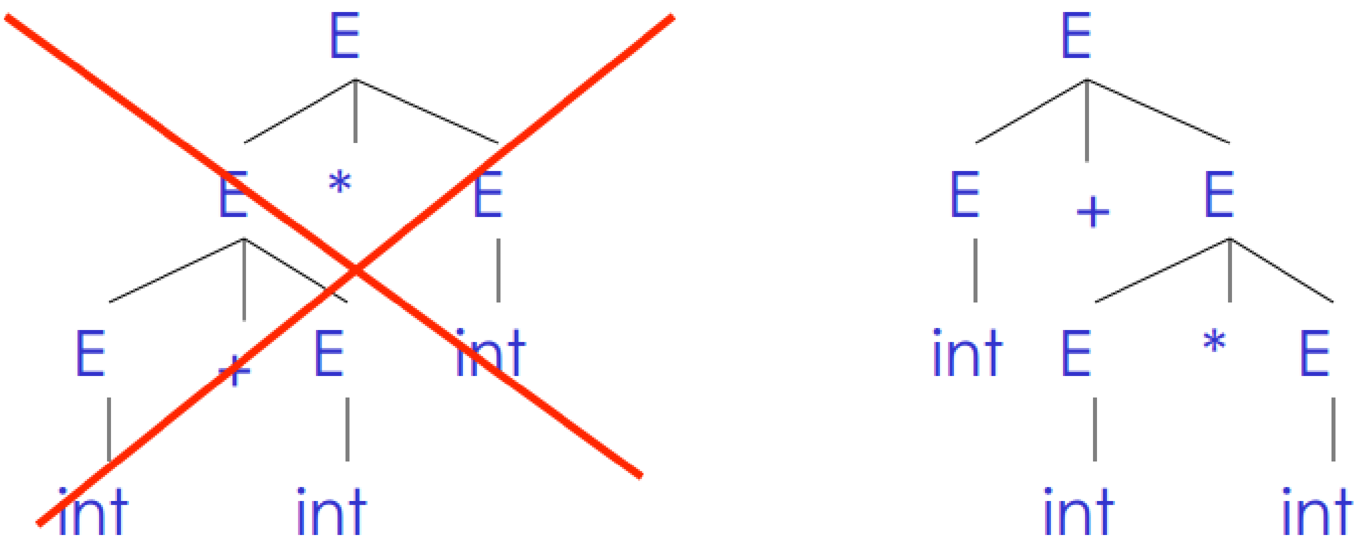




- Left associativity declaration: %left +

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int



- Precedence declarations: %left +
  %left *

A General Algorithm: Recursive Descent

- Let TOKEN be the type of all special tokens: INT, OPEN, CLOSE, PLUS, TIMES.
- Let the global variable next point to the next token.
- Define boolean functions that check for a match of
  - A given token terminal

```
boolean term(Token tok)
    {return  next++==tok;}
```

- The nth production of non-terminal S;

  boolean S_n(){…}

- Try all productions of S (which succeeds if any of the productions for S matches the input)

  boolean S(){…}

Example

$E \rightarrow T$

$E \rightarrow T+E$

$T \rightarrow int$

$T \rightarrow int*T$

$T \rightarrow (E)$

- To start the parser
  - Initialize next to point to first token
  - Invoke E()

- Easy to implement by hand.

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$

(int)

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E()  {TOKEN *save = next; return    (next = save, E1())
                                       || (next = save,  E2());   }
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next;  return   (next = save, T1())
                                       || (next = save,  T2())
                                       || (next = save,  T3()); }
```

Problem: Left Recursion

- Given a production S➔Sα

  boolean S1(){return S()&&term(α)

  boolean S(){return S1();}
- S() goes into an infinite loop.
- Because of the left recursion
- Recursive Descent does not work in such cases
- We need to eliminate left recursion

Eliminating Left Recursion

- Consider the grammar

  S→Sα|β
- Notice this grammar generates all strings starting with a β and followed by any number of α's
- To eliminate left recursion, we will rewrite using right recursion.
- We introduce a new non-terminal S', and write
- S→ßS'
- S'→αS'|ξ

In General

- $S \rightarrow S\alpha_1 | \ldots | S\alpha_n | ß_1 | \ldots | ß_m$
- All strings derived from S start with one of $ß_1$, …, $ß_m$ and continue with several instances of $\alpha_1 \ldots \alpha_n$.

- Rewrite as
- $S \rightarrow ß_1 S' | \ldots | ß_m S'$
- $S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \xi$

Predictive Parsing

- Like recursive descent but parser predict which production to use

  - Using look ahead (works with restricted grammar)
  - No backtracking
- Predictive parsers accept LL(K) grammars
  - Left to right
  - Left most derivation
  - K tokens look ahead (usually k=1)

LL(1)

- In LL(1)
  - At each step only one choice of production
  - Given wAb   on input t, there is at most one production that can be used

Refactoring

- Consider the grammar
- E➔T+E|T
- T➔int|int*T|(E)
- It is hard to predict which production to use
  - There are two production that can be used for E
  - and two productions that can be used for T (the two that begin with int)
  - This grammar is not acceptable for predictive LL(1) parsing

- We need to left-factor the grammar
- By eliminating common prefixes
- Example

  E ➔ T+E | T

- Becomes

  E ➔ TX

  X ➔ +E | ξ

- T➔int|int*T|(E)
- Becomes
  - T➔ int Y|(E)
  - Y➔*T|ξ

- What we did
  - We factored out the common prefix (which is T in the first example and int in the second)
  - We introduced a new nonterminal (X in the first example and Y in the second)
  - We used one production for T and
  - one for the new non-terminal that list all choices

Predictive Parsing

- Like recursive descent but parser predict which production to use
  - Using look ahead (works with restricted grammar)
  - No backtracking
- Predictive parsers accept LL(K) grammars
  - Left to right
  - Left most derivation
  - K tokens look ahead (usually k=1)

LL(1)

- In LL(1)
  - At each step only one choice of production
  - Given wAb   on input t, there is at most one production that can be used

Refactoring

- Consider the grammar
- E➔T+E|T
- T➔int|int*T|(E)
- It is hard to predict which production to use
  - There are two production that can be used for E
  - and two productions that can be used for T (the two that begin with int)
  - This grammar is not acceptable for predictive LL(1) parsing

- We need to left-factor the grammar
- By eliminating common prefixes
- Example

  E➔T+E|T

- Becomes

  E➔TX

  X➔+E|ξ

- T➔int|int*T|(E)
- Becomes
  - T➔ int Y|(E)
  - Y➔*T|ξ

- What we did
  - We factored out the common prefix (which is T in the first example and int in the second)
  - We introduced a new nonterminal (X in the first example and Y in the second)
  - We used one production for T and
  - one for the new non-terminal that list all choices

- Left factored grammar

  E → TX   X→ +E|ξ

  T → (E) |Y   Y → *T| ξ

- The LL(1) parsing table

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

- The leftmost column represents the leftmost. non-terminal symbol in a derivation
- The top row represents the next input token.
- For example the [E, int] entry, says
  - When current non-terminal is E and next input is int, use production E → TX

- Notice blank entries represent errors
- For example entry [E, *] is blank
- Indicating that there is no production to use for E to get successful parsing, in the input token is *.

LL(1) algorithm

- A method similar to recursive descent except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]
- Use a stack to record leaf nodes (frontiers) of the parse tree
- The top of stack is the leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input and empty stack

The LL(1) Algorithm

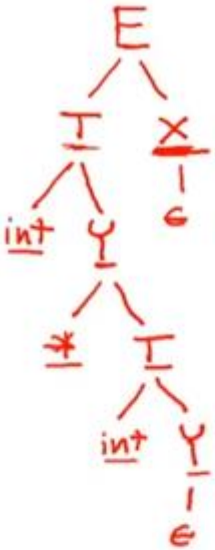
- Suppose a grammar has start symbol S and LL(1) parsing table T. We want to parse string ω
- Initialize a stack containing S$.
- Repeat until the stack is empty:
  - Let the next character of ω be t
  - If the top of the stack is a terminal r:
    - If r and t don't match, report an error.
    - Otherwise consume the character t and pop r from the stack.
  - Otherwise, the top of the stack is a nonterminal A:
    - If T[A, t] is undefined, report an error.
    - Replace the top of the stack with T[A, t].

Example

- Let's parse int*int, drawing the parse tree at each step.

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT



Constructing the Parse Table

- Consider
  - A non-terminal A
  - Production $A \rightarrow \alpha$
  - And an input token t
  - We want to know the conditions under which we can make the move $T[A,t]=\alpha$
- We make the move $T[A,t]=\alpha$ in two situations

1. If $\alpha \rightarrow^{*} t\beta$ i.e. α can derive a t in the first position

In this case we say that $t \in First(\alpha)$

And the move $T[A,t]=\alpha$ is reasonable

2. Or if $A \rightarrow \alpha$, and

   $\alpha \rightarrow^{*} \xi$ (i.e. α can disappear), and

   $S \rightarrow^{*} \beta\ A\ t\ \delta$ (notice since α can disappear so does A)

– Notice that this is useful if t can follow A and A can disappear.

– In other words A does not derive t but t follows A.

- This case we say t ϵ Follow(A)

First Sets

- Def.
- $First(X)=\{t \mid X \rightarrow^{*} t\alpha\} \vee \{ \xi \mid X \rightarrow^{*} \xi \}$
- Notice that the last part is there because we need to keep track of whether or not X can produce ξ.
- Algorithm :

1. If t is a terminal

   First (t) = { t }

2. If X is non-terminal, then $\xi \in$ First(X)
   1. If $X \rightarrow \xi$
   2. Or if $X \rightarrow A_1, \ldots A_n$ and $\xi \in$ First(A_i) for $1 \leq i \leq n$
      i.e. if $A_1, \ldots A_n$ can disappear by producing ξ
3. First (α) is a subset of First(X) if

   $X \rightarrow A_1, \ldots A_n \alpha$

   and $\xi \in$ First(A_i) for $1 \leq i \leq n$

   (i.e. $A_1, \ldots A_n$ can all disappear)

Example on First Sets

- E → T X
- T → (E) | int Y

X → +E | ξ

Y → * T| ξ

1. Terminals

First(+)={+}

First(*)={*}

First(()={(}

First())={)}

First(int)={int}

2. Non-terminals
   - First(E)
     1. Since E → TX , then First(E) is a super set of First(T) and First(T) = { (, int }
     2. Notice if T →* ξ then First(E) is a super set of First(X) but this is not the case since First(T) does not contain ξ

     Therefore, First(E) = First(T)= { (, int }
   - First(X)= { + , ξ }
   - First(Y)= { * , ξ }

Follow Sets

- Notice Follow(X) is not about what X produces but rather about where X appears.
- Definition

  $$\text{Follow}(X) = \{\, t \mid S \rightarrow^{*} \beta\, X\, t\, \delta \,\}$$

- Intuition
  - If $X \rightarrow A\beta$ then
    - First(B) is a subset of Follow(A)
    - Follow(X) is a subset of Follow(B) (i.e., anything that can come after X is included in the follow of B)

- If $X \rightarrow A\beta$ and $\beta \rightarrow^{*} \xi$

  then Follow(X) is a subset of Follow(A)

  (i.e., anything that can come after X is included in Follow(A))

- If S is the start symbol, then $ ϵ Follow(S)

  (we always add $ in the Follow of the start symbol)

  Because it is what we have when we runout of input)

Algorithm

1. $ ϵ Follow(S), where S is the start symbol
2. For each production A → α X β

   First(β) – { ξ } is a subset of Follow(X)

   (notice that we exclude ξ , because ξ is never in a follow set)
3. For each production A → α X β

   if ξ ϵ First(β) (i.e., β can completely disappear)

   then whatever is in Follow(A) is also in Follow(X)

   i.e., Follow(A) is a subset of Follow(X)

Example

- $E \rightarrow T\ X$
- $T \rightarrow (E)\ |\ int\ Y$

$X \rightarrow +E\ |\ \xi$

$Y \rightarrow *\ T |\ \xi$

- Remember to determine the follow of X we need to look at where X appears
- Follow(E)
  1. Since E is a start symbol, \$ is ϵ Follow(E)
  2. Since $T \rightarrow (E)$, then) is ϵ Follow(E)
  3. Since $X \rightarrow +E$, then anything that is in the follow of X is also in the follow of E (i.e. Follow(X) is a subset of Follow(E))
  4. Since $E \rightarrow T\ X$ then any thing that is in the follow of E is also in the follow of X (i.e. Follow(E) is a subset of Follow(X))
  5. From 3 and 4 we conclude that Follow(E)=Follow(X)
  6. Both are { \$,) }

- Follow(T)
  1. Since E → T X, then Follow(T) includes First(X) (which is {+, ξ} but we must exclude ξ).
  2. Since X → ξ , Follow(T) must include follow(E)
  3. (i.e. Follow(E) is a subset of Follow(T))
  4. Since T also appears in Y → * T then Follow(T) includes Follow(Y) (Follow(Y) is a subset of Follow(T)
  5. But notice that T → int Y so Follow(T) is also a subset of Follow(Y)
  6. From 4 and 5, we conclude that Follow(T)=Follow(Y)={ +, $,) }

Follow of Terminal Symbols

- Follow(‘(‘)
  - Since ‘(‘ appears in T → (E), then Follow(‘(‘) includes First(E) (i.e. it includes { (, int })
  - Since ‘(‘ does not appear anywhere else
  - Follow(‘(‘)= { (, int }

- Follow(‘)’)
  - Since ‘)’ appears only in T → (E),

    Follow(‘)’) must include only Follow(T)
  - Follow(‘)’) = {+, $,)}

- Follow('+')
  - Since + is only used in X → +E

    Follow('+') includes First(E), which is { (, int} .
  - Notice the E cannot produce ξ
  - Follow('+') = { (, int}

- Follow(‘*’)
  - Since ‘*’ is only used in Y → * T

    Follow(‘*’)   includes First(T), which is { (, int}
  - Since T cannot got to ξ then that is it
  - Follow(‘*’)= { (, int}

- Follow(int)
  - Since int only appears in T → int Y
  - Follow(int) includes First(Y) which is {*}
  - But since Y→ξ, Y can completely diappear therefore, Follow(int) must include Follow(T) (which is {+,$,)})
  - Follow(int)={*, +,$,)}

Putting Together First sets and Follow Sets to Construct an LL(1) table

- For each production A→ α in G do
  - For each terminal t ϵ First(α) do
    - T[A,t]= α because obviously would is useful here
  - If ξ ϵ First(α), for each t ϵ Follow(A) do
    - T[A,t]= α because α can completely disappear and consequently A disappears.
  - If ξ ϵ First(α) and $ ϵ Follow(A) do
    - T[A,$]= α This is useful when we ran out of input because the only hope would be is to get rid of whatever is on the stack.

Example

- E → T X
- T → (E) | int Y

X → +E | ξ

Y → * T| ξ

	(	)	+	*	int	$
E	TX				TX	
T	(E)				int Y	
X		ξ	+E			ξ
Y		ξ	ξ	*T		ξ

T[E,( ] = T[E, int] = TX because (and int are in the First of TX
T[T, ( ] = (E) because (is in the First((E))
T[T, int] = int Y because int is in the First(int Y)
T[X, +] = +E because + is in the First(+E)
T[Y, *] = *T because * is in the First(*T)
T[X,)] = ξ because X → ξ and) is in the Follow(X)
T[X,$]= ξ because X → ξ and $ is in the Follow(X)
T[Y,)] = ξ because Y → ξ and) is in the follow(Y)
T[Y, +] = ξ because Y → ξ and + is in the follow(Y)
T[Y, $] = ξ because Y → ξ and $ is in the follow(Y)

Not all grammars are LL(1) grammars

- Example:
- S→ Sa| b
- First(S) = {b}
- Follow(S) = { $, a}
- Let's try to construct an LL(1) table

	a	**b**	**$**
S		b Sa	

- Notice that we have multiply defined entry
- i.e., 2 possible moves to make, not deterministic
- We conclude that the grammar is not LL(1) grammar

- If an entry is multiply defined, the G is not an LL(1) grammar
- The list includes (but not limited to)
  - Any grammar that is not left factored
  - Any grammar that contains left recursion (the above example)
  - Any grammar that is ambiguous
  - Any grammar that requires more than 1 look ahead token
- Remember the above list is not comprehensive
- The only way to make sure is by trying to construct an LL(1) parsing table

- Most programming languages CFGs are not LL(1).
- LL(1) grammars are to weak to capture many interesting constructs in PLs
- The solution will build up on what we have learned so far.