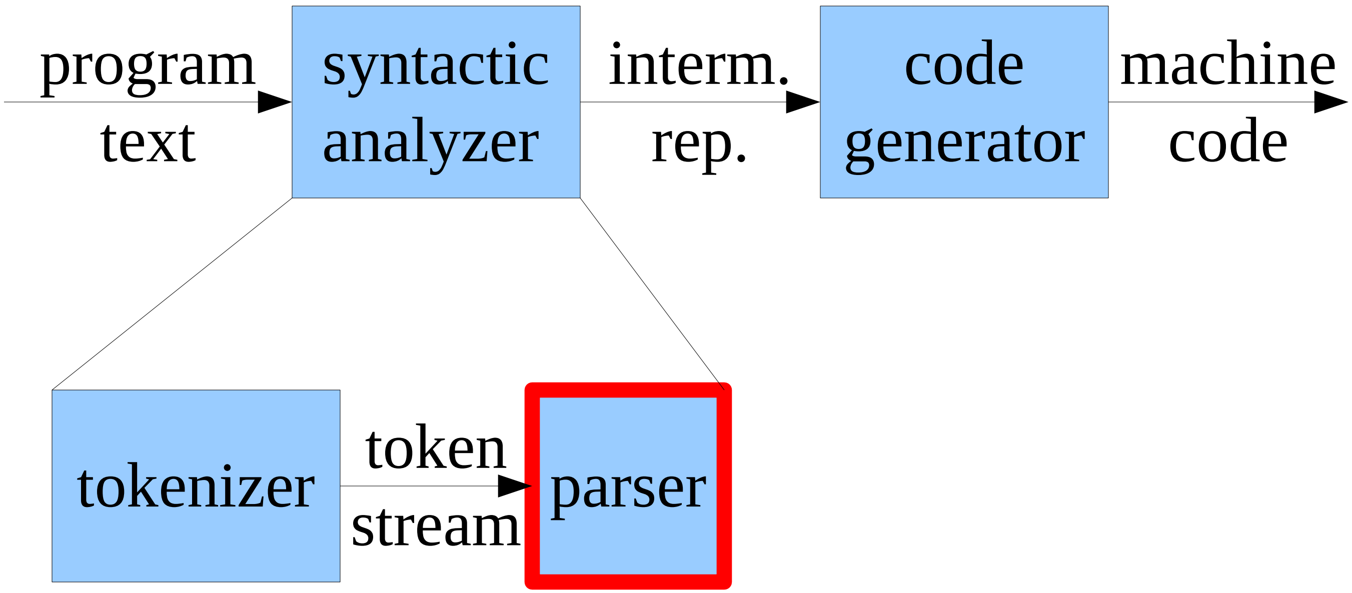
Parsing

COMP 3002

School of Computer Science

Carleton University
Canada's Capital University

The Structure of a Compiler

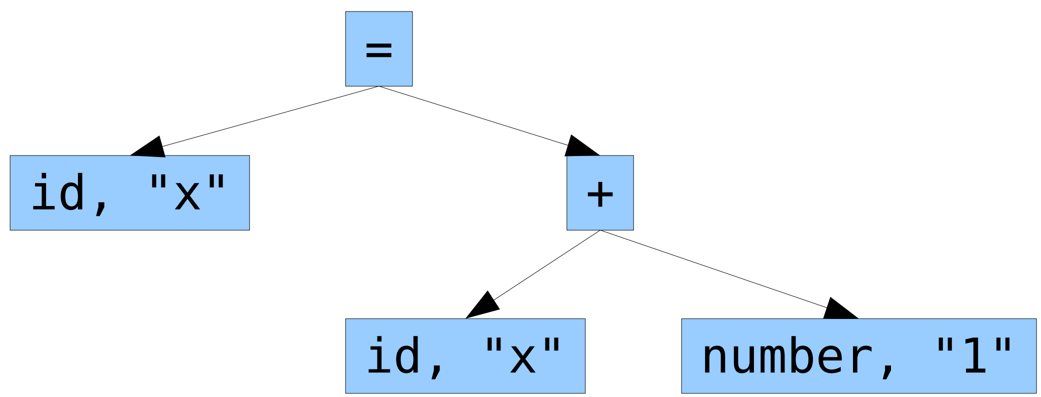


Role of the Parser

- Converts a token stream into an intermediate representation
  - Captures the *meaning* (instead of text) of the program
  - Usually, intermediate representation is a *parse tree*

```
x = x + 1
```

```
<id, "x"> <assign> <id, "x"> <plus> <number, "1">
```



Kinds of Parsers

- Universal
  - Can parse any grammar
  - Cocke-Younger-Kasami and Earley's algorithms
  - Not efficient enough to be used in compilers
- Top-down
  - Builds parse trees from the top (root) down
- Bottom-up
  - Builds parse trees from the bottom (leaves) up

Carleton UNIVERSITY
Canada's Capital University

Errors in Parsing

- Lexical errors
  - Misspelled identifiers, keywords, or operators
- Syntactical errors
  - Misplaced or mismatched parentheses, case statement outside of any `switch` statement,…
- Semantic errors
  - Type mismatches between operators and operands
- Logical errors
  - Bugs – the programmer said one thing but meant something else
    - `if (x = y) { ... }`
    - `if (x == y) { ... }`

Carleton UNIVERSITY
Canada's Capital University

Error Reporting

- A parser should
  - report the presence of errors clearly and correctly and
  - recover from errors quickly enough to detect further errors

Error Recovery Modes

- Panic-Mode
  - discard input symbols until a "synchronizing token" is found
  - Examples (in Java): semicolon, '}'
- Phrase-Level
  - replace a prefix of the remaining input to correct it
  - Example: Insert ';' or '{'
  - must be careful to avoid infinite loops

Carleton UNIVERSITY
Canada's Capital University

Error Recover Modes (Cont'd)

- Error Productions
  - Specify common errors as part of the language specification
- Global Correction
  - Compute the smallest set of changes that will make the program syntactically correct (impractical and usually not usually useful)

Carleton UNIVERSITY
Canada's Capital University

Context-Free Grammars

- CF grammars are used to define languages
- Specified using BNF notation
  - A set of non-terminals N
  - A set of terminals T
  - A list of rewrite rules (*productions*)
  - The LHS of each rule contains one non-terminal symbol
  - The RHS of each rule contains a regular expression over the alphabet $N \cup T$
  - A special non-terminal is usually designated as the start symbol
    - Usually, start symbols is LHS of the first production

Carleton UNIVERSITY
Canada's Capital University

Context Free Grammars and Compilers

- In a compiler
  - *N* consists of language constructs (function, block, if-statement, expression, ...)
  - *T* consists of tokens

Carleton UNIVERSITY
Canada's Capital University

Grammar Example

- Non-terminals: E, T, F
  - E = expression
  - T = term
  - F = factor
- Terminals: **id**, +, *, (,)
- Start symbol E
- "Mathematical formulae using + and *

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

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

$$\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}$$

Carleton University

Canada's Capital University

Derivations

- From a grammar specification, we can derive any string in the language
  - Start with the start symbol
  - While the current string contains some non-terminal *N*
    - expand *N* using a rewrite rule with *N* on the LHS

Carleton UNIVERSITY
Canada's Capital University

```
E → E + E | E * E | ( E ) | id
```

```
E
E + E
E + id
E * E + id
E * id + id
( E ) * id + id
( E + E ) * id + id
( id + E ) * id + id
( id + id ) * id + id
```

Derivation Example

- Derive `id + id * id` with these grammars:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$$

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

$$\begin{aligned} E &\rightarrow T\ E' \\ E' &\rightarrow +\ T\ E' \mid \varepsilon \\ T &\rightarrow F\ T' \\ T' &\rightarrow *\ F\ T' \mid \varepsilon \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

Carleton UNIVERSITY
Canada's Capital University

Derivation Example

- Derive:
  - `id * id + id`
  - `id + id * id`

```
E  → T E'
E' → + T E' | ε
T  → F T'
T' → * F T' | ε
T → T * F | F
F → ( E ) | id
```

Carleton UNIVERSITY
Canada's Capital University

Terminology

- The strings of terminals that we derive from the start symbol are called *sentences*
- The strings of terminals and non-terminals are called *sentential forms*

Carleton UNIVERSITY
Canada's Capital University

Leftmost and Rightmost Derivations

- A derivation is *leftmost* if at each stage we always expand the leftmost non-terminal
- A derivation is *rightmost* if at each stage we always expand the rightmost non-terminal
- Give a leftmost and rightmost derivation of
  - id * id + id

$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$

Derivations and Parse Trees

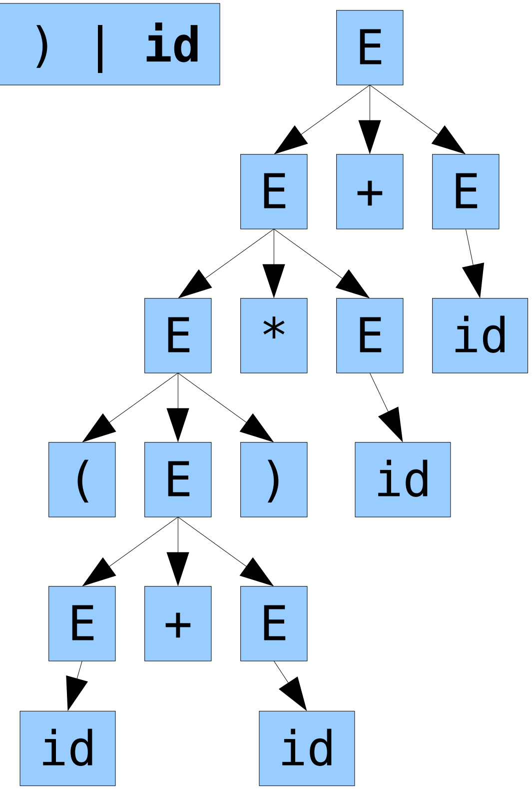
- A *parse-tree* is a graphical representation of a derivation
- Internal nodes are labelled with non-terminals
  - Root is the start symbol
- Leaves are labelled with terminals
  - String is represented by left-to-right traversal of leaves
- When applying an expansion `E` $\rightarrow$ `ABC...Z`
  - Children of node E become nodes labelled `A,B,C,...Z`

Carleton UNIVERSITY
Canada's Capital University

Derivations and Parse Trees - Example

E → E + E | E * E | (E) | **id**

E
E + E
E + **id**
E * E + **id**
E * **id** + **id**
(E) * **id** + **id**
(E + E) * **id** + **id**
(id + E) * **id** + **id**
(id + id) * **id** + **id**

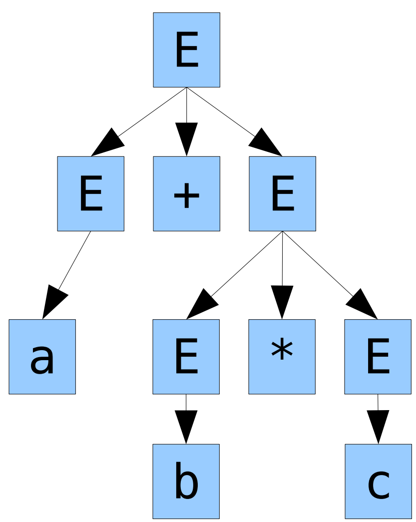


Ambiguity

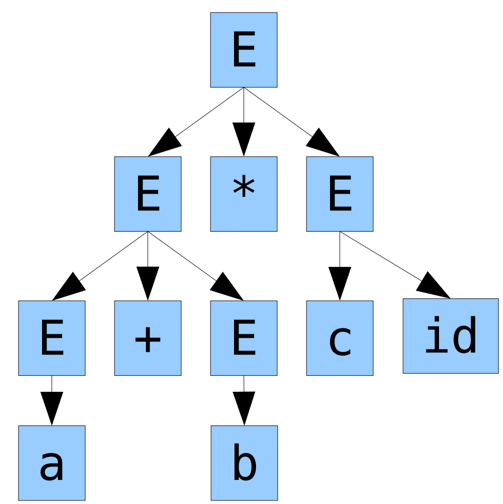
$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$

- Different parse trees for the same sentence result in ambiguity

```
E
E + E
E + E * E
a + b * c
```




```
E
E * E
E + E * E
a + b * c
```



Ambiguity – Cont'd

- Ambiguity is usually bad
- The same program means two different things
- We try to write grammars that avoid ambiguity

Carleton UNIVERSITY

Canada's Capital University

Context Free Grammars and Regular Expressions

- CFGs are more powerful than regular expressions
  - Converting a regular expression to a CFG is trivial
  - The CFG $S \to aSb \mid \varepsilon$ generates a language that is not regular
- But not that powerful
  - The language $\{ a^m b^n c^m d^n : n,m>0 \}$ can not be expressed by a CFG

Carleton UNIVERSITY
Canada's Capital University

Enforcing Order of Operations

- We can write a CFG to enforce specific order of operations
  - Example: + and *
  - Exercises:
    - Add comparison operator with lower level of precedence than +
    - Add exponentiation operator with higher level of precedence than *

```
 E  → PE
PE  → TE + PE | TE
TE  → id * TE | id | ( E )
```

Carleton UNIVERSITY
Canada's Capital University

Picking Up - Context free grammars

- CFGs can specify programming languages
- It's not enough to write a correct CFG
  - An ambiguous CFG can give two different parse trees for the same string
    - Same program has two different meanings!
  - Not all CFGs are easy to parse efficiently
- We look at restricted classes of CFGs
  - Sufficiently restricted grammars can be parsed easily
  - The parser can be generated automatically

Parser Generators

- Benefits of parser generators
  - No need to write code (just grammar)
  - Parser always corresponds exactly to the grammar specification
  - Can check for errors or ambiguities in grammars
  - No surprise programs
- Drawbacks
  - Need to write a restricted class of grammar [LL(1), LR(1), LR(k),...]
  - Must be able to understand when and why a grammar is not LL(1) or LR(1) or LR(k)
  - Means learning a bit of parsing theory
  - Means learning how to make your grammar LL(1), LR(1), or LR(k)

Carleton UNIVERSITY
Canada's Capital University

Ambiguity

- This grammar is ambiguous
  - consider the input
    - **`a – b – c`**

$$E \rightarrow E \texttt{ - } E \mid \textbf{id}$$

- Rewrite this grammar to be unambiguous
- Rewrite this grammar so that - becomes left associative:
    - **`a – b – c ~ ((a – b) – c)`**

Carleton UNIVERSITY
Canada's Capital University

Solutions

```
E → id M
M → - E | ε
```

```
E → M | id
M → E - id
```

Carleton UNIVERSITY
Canada's Capital University

A Common Ambiguity – The Dangling Else

```
stmt → if expr then stmt
     |  if expr then stmt else stmt
     |  other
```

- Show that this grammar is ambiguous
- Remove the ambiguity
  - Implement the "**else** matches innermost **if**" rule

Carleton UNIVERSITY
Canada's Capital University

Solution

```
stmt         → matched_stmt | open_stmt
matched_stmt → if expr then matched_stmt
                else matched_stmt
              | other
open_stmt    → if expr then stmt
              | if expr then matched_stmt
                else open_stmt
```

Carleton University

Canada's Capital University

Left Recursion

- A top-down parser expands the left-most non-terminal based on the next token
- Left-recursion is difficult for top-down parsing
- Immediate left recursion:
  - $A \rightarrow A\alpha \mid \beta$
  - Rewrite as: $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$
- More complicated left recursion occurs when A can derive a string starting with A
  - $A \rightarrow^{+} A\alpha$

Carleton UNIVERSITY
Canada's Capital University

Removing Left Recursion

- Removing immediate left-recursion is easy
- Simple case:
  - $A \rightarrow A\alpha \mid \beta$
  - Rewrite as: $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \varepsilon$
- More complicated:
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_\kappa \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_\tau$
  - Rewrite as:
    - $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_\tau A'$
    - $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_\kappa A' \mid \varepsilon$

Carleton UNIVERSITY
Canada's Capital University

Algorithm for Removing all Left Recursion

- Textbook page 213

Carleton UNIVERSITY

Canada's Capital University

Left Factoring

- Left factoring is a technique for making a grammar suitable for top-down parsing
- For each non-terminal *A* find the longest prefix α common to two or more alternatives
  - Replace $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n$ with
  - $A \rightarrow \alpha A'$ and $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$
- Repeat until no two alternatives have a common prefix

Carleton UNIVERSITY
Canada's Capital University

Left Factoring Example

- Left factor the following grammars

```
 E  → PE
PE  → TE + PE | TE
TE  → id * TE | id | ( E )
```

Carleton UNIVERSITY
Canada's Capital University

Summary of Grammar-Manipulation Tricks

- Eliminating ambiguity
  - Different parse trees for same program
- Enforcing order of operations
  - Left-associative
  - right-associative
- Eliminating left-recursion
  - Gets rid of potential "infinite recursions"
- Left factoring
  - Allows choosing between alternative productions based on current input symbol

Carleton UNIVERSITY
Canada's Capital University

Exercise

$$
\begin{array}{lll}
\texttt{rexpr} & \rightarrow & \texttt{rexpr + rterm | rterm} \\
\texttt{rterm} & \rightarrow & \texttt{rterm rfactor | rfactor} \\
\texttt{rfactor} & \rightarrow & \texttt{rfactor * | rprimary} \\
\texttt{rprimary} & \rightarrow & \mathbf{a} \texttt{ | } \mathbf{b}
\end{array}
$$

- Remove left recursion
- Left-factor

Carleton UNIVERSITY
Canada's Capital University

Top-Down Parsing

- Top-down parsing is the problem of constructing a pre-order traversal of the parse tree
- This results in a leftmost derivation
- The expansion of the leftmost non-terminal is determined by looking at a prefix of the input

Carleton UNIVERSITY
Canada's Capital University

LL(1) and LL(k)

- If the correct expansion can always be determined by looking ahead at most k symbols then the grammar is an LL(*k*) grammar
- LL(1) grammars are most common

Carleton UNIVERSITY
Canada's Capital University

FIRST(α)

- Let α be any string of grammar symbols
- FIRST(α) is the set of terminals that begin strings that can be derived from α a
  - If α can derive ε then ε is also in FIRST(α)
- Why is FIRST useful
  - Suppose $A \rightarrow \alpha \mid \beta$ and FIRST(α) and FIRST(β) are disjoint
  - Then, by looking at the next symbol we know which production to use next

Carleton UNIVERSITY
Canada's Capital University

Computing FIRST(X)

- If X is a terminal then FIRST(X) = {X}
- If X is a non-terminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$
  - $i = 0$; define $\mathrm{FIRST}(Y_0) = \{\ \varepsilon\ \}$
  - while ε is in $\mathrm{FIRST}(Y_i)$
    - Add $\mathrm{FIRST}(Y_{i+1})$ to FIRST(X)
    - $i = i+1$
  - if ($i = k$ or $X \rightarrow \varepsilon$)
    - Add ε to FIRST(X)
- Repeat above step for all non-terminals until nothing is added to any FIRST set

Carleton UNIVERSITY
Canada's Capital University

Carleton UNIVERSITY
Canada's Capital University

Example

- Compute FIRST(E), FIRST(PE), FIRST(TE), FIRST(TE')

```
 E   → PE
PE   → TE + E | TE
TE   → id TE'
TE'  → * E
TE'  → id
TE'  → ( E )
```

Computing FIRST($X_1X_2...X_k$)

- Given FIRST(X) for every symbol X we can compute FIRST($X_1X_2...X_k$) for any string of symbols $X_1X_2...X_k$:
  - i = 0 ; define FIRST(X_0) = { ε }
  - while ε is in FIRST(X_i)
    - Add FIRST(X_{i+1}) to FIRST($X_1X_2...X_k$)
    - i = i+1
  - if (i =k)
    - Add ε to FIRST(X)

Carleton UNIVERSITY
Canada's Capital University

FOLLOW(A)

- Let A be any non-terminal
- FOLLOW(A) is the set of terminals **a** that can appear immediately to the right of A in some sentential form
  - I.e. $S \rightarrow^{*} \alpha A \text{ a } \beta$ for some α and β and start symbol S
  - Also, if A can be a rightmost symbol in some sentential form then \$ (end of input marker) is in FOLLOW(A)

Carleton UNIVERSITY
Canada's Capital University

Computing FOLLOW(A)

- Place \$ into FOLLOW(S)
- Repeat until nothing changes:
  - if $A \rightarrow \alpha B\beta$ then add $FIRST(\beta)\setminus\{\varepsilon\}$ to FOLLOW(B)
  - if $A \rightarrow \alpha B$ then add FOLLOW(A) to FOLLOW(B)
  - if $A \rightarrow \alpha B\beta$ and ε is in $FIRST(\beta)$ then add FOLLOW(A) to FOLLOW(B)

Carleton UNIVERSITY
Canada's Capital University

Example

- Compute FOLLOW(E), FOLLOW(PE), FOLLOW(TE), FOLLOW(TE')

```
 E   → PE
PE   → TE + E | TE
TE   → id TE'
TE'  → * E
TE'  → id
TE'  → ( E )
```

FIRST and FOLLOW Example

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \textbf{id}
\end{aligned}
$$

- FIRST(F) = FIRST(T) = FIRST(E) = {(, id }
- FIRST(E') = {+, ε}
- FIRST(T') = {*, ε}
- FOLLOW(E) = FOLLOW(E') = {), $}
- FOLLOW(T) = FOLLOW(T') = {+,),$}
- FOLLOW(F) = {+, *,), $}

Carleton UNIVERSITY
Canada's Capital University

LL(1) Grammars

- Left to right parsers producing a leftmost derivation looking ahead by at most 1 symbol
- Grammar G is LL(1) iff for every two productions of the form $A \rightarrow \alpha \mid \beta$
  - FIRST(α) and FIRST(β) are disjoint
  - If ε is in FIRST(β) then FIRST(α) and FOLLOW(A) are disjoint (and vice versa)
- Most programming language constructs are LL(1) but careful grammar writing is required

Carleton UNIVERSITY
Canada's Capital University

LL(1) Predictions Tables

- LL(1) languages can be parsed efficiently through the use of a prediction table
  - Rows are non-terminals
  - Columns are input symbols (terminals)
  - Values are productions

Constructin LL(1) Prediction Table

- The following algorithm constructs the LL(1) prediction table
- For each production $A \rightarrow \alpha$ in the grammar
  - For each terminal a in FIRST(α), set M[A,a] = $A \rightarrow \alpha$
  - If ε is in FIRST(α) then for each terminal b in FOLLOW(A), set M[A,b] = $A \rightarrow \alpha$

Carleton UNIVERSITY
Canada's Capital University

LL(1) Prediction Table Example

	Id	+	*	(	)	$
E	E $\rightarrow$ T E'			E $\rightarrow$ T E'		
E'		E' $\rightarrow$ + T E'			E' □ e	E' $\rightarrow$ ε
T	T $\rightarrow$ F T'			T $\rightarrow$ F T'		
T'		T' $\rightarrow$ ε	T' $\rightarrow$ * F T'		T' $\rightarrow$ ε	T' $\rightarrow$ ε
F	F $\rightarrow$ id			F $\rightarrow$ (E)		

E $\rightarrow$ T E'
E' $\rightarrow$ + T E' | ε
T $\rightarrow$ F T'
T' $\rightarrow$ * F T' | ε
F $\rightarrow$ (E) | **id**

LL(1) Prediction Table Example

- Use the table to find the derivation of
  - id + id * id + id

	Id	+	*	(	)	$
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E'		$E' \rightarrow +\ T\ E'$			E' □ e	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *\ F\ T'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (\ E\ )$		

LL(1) Parser Generators

- Given a grammar *G*, an LL(1) parser generator can
  - Computer FIRST(*A*) and FOLLOW(*A*) for every non-terminal *A* in *G*
  - Determine if *G* is LL(1)
  - Construct the prediction table for *G*
  - Create code that parses any string in *G* and produces the parse tree
- In Assignment 2 we will use such a parser generator (javacc)

Carleton University
Canada's Capital University

Summary

- Programming languages can be specified with context-free grammars
- Some of these grammars are easy to parse and generate a unique parse tree for any program
- An LL(1) grammar is one for which a leftmost derivation can be done with only one symbol of lookahead
- LL(1) parser generators exist and can produce efficient parsers given only the grammar

Carleton UNIVERSITY
Canada's Capital University