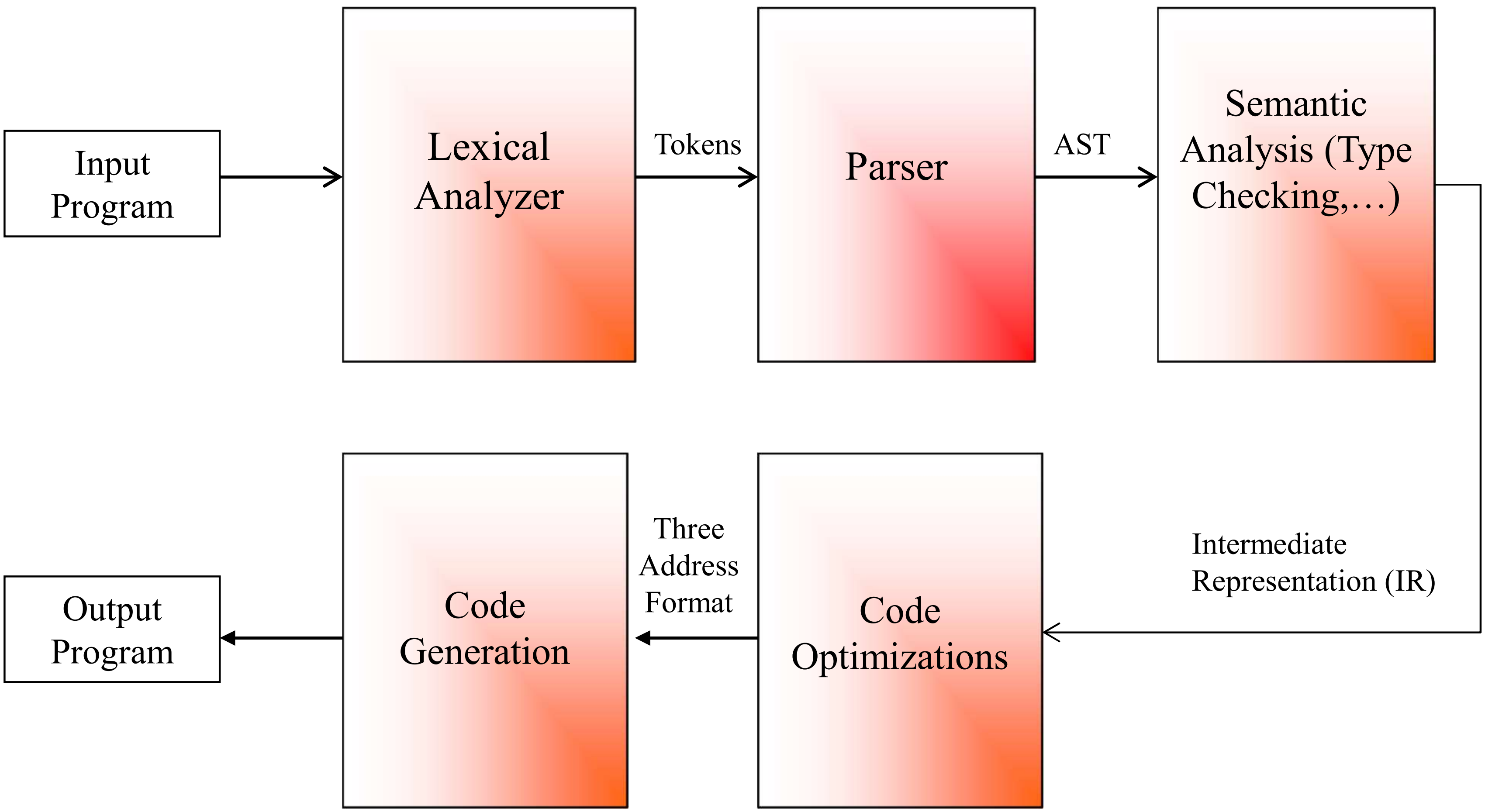
Top-down Parsing: Predictive and LL(k) parsers

The Structure (Phases) of a Compiler



Bottom-up vs. Top-down

	Top-down Parsers	Bottom-up Parsers
Successful Parse	From start symbol of grammar to the string	From the string to the start symbol of the grammar
Example of grammars	LL(k) Left-to-right, Leftmost derivation first	LR(k), LALR Left-to-right, Rightmost derivation first (in reverse)
Example of parser technique	Recursive-descent	Shift-reduce
Ease of implementation	Very easy	Many grammar generators available (Yacc, Bison,…)
Issue with left-recursion	Yes for recursive-descent	No
Issues with left-factoring	Yes for recursive-descent	No

Summary of Recursive Descent Parsers

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically

- Unpopular because of backtracking
  - Inefficient

- Inefficiency of recursive-descent is the primary motivation for developing more complex parsers

Predictive Recursive-descent Parser vs. LL(k) Parsers

- Predictive Recursive-descent Parser
  - –Typically hand written
  - –Follow the recursive-descent template
  - –For every LL(k) grammar there is a corresponding predictive parser
- LL(k) Parsers
  - –Table driven
  - –Generated using parser generators
  - –For every LL(k) grammar there is a corresponding LL(k) parser

Terminology: Difference between LL(k) Parsers and LL(k) Grammars

- First, what is the difference between CFG, CFL and Parsers?
  - Context-free Language (CFL): A set of strings
  - Context-free Grammar (CFG): A compact representation
  - Parser: A program that accepts strings and returns parse trees

- An LL(k) Grammar is a CFG whose strings can be parsed by a LL(k) table-driven parser or a predictive LL(k) recursive-descent parser

LL(k) Table-driven Parsers

LL(k) Table-driven Parsers

- Input to the parser: string to be parsed
- Output of the parser: parse tree

- Generated using parser generator
- Table-driven
- Predictive, i.e., given a non-terminal to expand and an input token, predicts which rule may lead to a parse

LL(k) Parsers: Data Structures for the General Alogrithm

- The parser consists of
  - an input buffer that holds the input string
  - a stack on which to store the terminals and non-terminals yet to be parsed
  - a parsing table which tells it which grammar rule to apply given the symbols on top of its stack and the next input token

LL(1) Parsing and Left Factoring

- Recall the grammar

$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Hard to predict because
  - For T two productions start with int
  - Similarly, for E it is not clear how to predict

- We need to left-factor the grammar

Left-Factoring Example

- Recall the grammar

$$E \to T + E \mid T$$
$$T \to int \ \mid int * T \mid (E)$$

- Factor out common prefixes of productions

$$E \to T\ X$$
$$X \to + E \mid \varepsilon$$
$$T \to (E) \mid int\ Y$$
$$Y \to * T \mid \varepsilon$$

LL(1) Parsing Table Example

- Left-factored grammar

  $E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

  $T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- The LL(1) parsing table:

next input token

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

leftmost non-terminal

rhs of production to use

Using Parsing Tables

- 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]

- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to be matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on end of input & empty stack

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \to T\ X$"
  - This can generate an int in the first position

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

LL(1) Parsing Table Example (Cont.)

- Consider the [Y,+] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only if $Y \rightarrow \varepsilon$

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

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations

- Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Error

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

Using Parsing Tables

- 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]

- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to be matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

- Reject on reaching error state
- Accept on end of input & empty stack

LL(1) Parsing Algorithm

initialize stack = <S $> and next pointer to point
to leftmost symbol of the input
repeat
case top of stack, rest
<X, rest> : if T[X,*next] = $Y_1 \ldots Y_n$
then stack ← <$Y_1 \ldots Y_n$ rest>;
else error ();
<t, rest> : if t == *next ++
then stack ← <rest>;
else error ();
until stack == < >

LL(1) Parsing Algorithm

$ marks bottom of stack

```
initialize stack = <S $> and next
repeat
   case top of stack, rest
      <X, rest>   : if T[X,*next] = Y1...Yn
                       then stack ← <Y1... Yn rest>;
                       else  error ();
      <t, rest>   : if t == *next ++
                       then  stack ← <rest>;
                       else error ();
until stack == < >
```

For non-terminal X on top of stack, lookup production

Pop X, push production rhs on stack. Note leftmost symbol of rhs is on top of the stack.

For terminal t on top of stack, check t matches next input token.

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 Parsing Tables: The Intuition

- Consider non-terminal A, production $A \rightarrow \alpha$, & token t
- Under what conditions can we replace A with α, given A is at the top of the stack and next token is t?

- $T[A,t] = \alpha$ in two cases:

- If $\alpha \rightarrow^* t\ \beta$
  - α can derive a t in the first position
  - We say that $t \in First(\alpha)$

- If $A \rightarrow \alpha$ and $\alpha \rightarrow^* \varepsilon$ and $S \rightarrow^* \beta\ A\ t\ \delta$
  - Useful if stack has A, input is t, and A cannot derive t
  - In this case only option is to get rid of A (by deriving ε)
    - Can work only if t can follow A in at least one derivation
  - We say $t \in Follow(A)$

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

Computing First Sets

Definition

$$\text{First}(X) = \{\, t \mid X \to^{*} t\alpha \} \cup \{\varepsilon \mid X \to^{*} \varepsilon\}$$

Algorithm sketch:

1. $\text{First}(t) = \{\, t \,\}$
2. $\varepsilon \in \text{First}(X)$
   - if $X \to \varepsilon$
   - if $X \to A_1 \ldots A_n$ and $\varepsilon \in \text{First}(A_i)$ for all $1 \le i \le n$
3. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \to A_1 \ldots A_n\, \alpha$
   - and $\varepsilon \in \text{First}(A_i)$ for all $1 \le i \le n$

First Sets. Example

- Recall the grammar

$$E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$$
$$T \to (\,E\,) \mid \text{int}\ Y \qquad\qquad Y \to *\,T \mid \varepsilon$$

- First sets

First(( ) = { (}   First(T) = {int, (}

First( )) = {) }   First(E) = {int, (}

First(int) = { int }   First(X) = {+, ε }

First(+) = { + }   First(Y) = {*, ε }

First(*) = { * }

Computing Follow Sets

- Definition:

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

- Intuition
  - If $X \to A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and
    $\text{Follow}(X) \subseteq \text{Follow}(B)$
    - if $B \to^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
  - If S is the start symbol then $\$ \in \text{Follow}(S)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   – For each production $A \rightarrow \alpha\ X\ \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   – For each production $A \rightarrow \alpha\ X\ \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets. Example

- Recall the grammar

$E \to T\,X \qquad\qquad X \to +\,E \mid \varepsilon$

$T \to (\,E\,) \mid \text{int}\;Y \qquad\qquad Y \to *\;T \mid \varepsilon$

- Follow sets

Follow(+) = { int, (}   Follow(*) = { int, (}

Follow(( ) = { int, (}   Follow(E) = {), $}

Follow(X) = {$,) }   Follow(T) = {+,) , $}

Follow( )) = {+,) , $}   Follow(Y) = {+,) , $}

Follow(int) = {*, +,) , $}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

First Sets. Example

- Recall the grammar

$$E \to T\ X \qquad\qquad X \to +\ E \mid \varepsilon$$
$$T \to (\ E\ ) \mid \text{int}\ Y \qquad\qquad Y \to *\ T \mid \varepsilon$$

- First sets

First(( ) = { (} First(T) = {int, (}
First( )) = {) } First(E) = {int, (}
First(int) = { int } First(X) = {+, ε }
First(+) = { + } First(Y) = {*, ε }
First(*) = { * }

Follow Sets. Example

- Recall the grammar

$E \rightarrow T\,X \qquad\qquad X \rightarrow +\,E \mid \varepsilon$

$T \rightarrow (\,E\,) \mid \text{int}\;Y \qquad Y \rightarrow *\,T \mid \varepsilon$

- Follow sets

Follow(+) = { int, (} Follow(*) = { int, (}

Follow(( ) = { int, (} Follow(E) = {), $}

Follow(X) = {$,) } Follow(T) = {+,) , $}

Follow( )) = {+,) , $} Follow(Y) = {+,) , $}

Follow(int) = {*, +,) , $}

LL(1) Parsing Table Example

- Left-factored grammar

  $E \rightarrow T\ X$ $\qquad$ $X \rightarrow +\ E \mid \varepsilon$

  $T \rightarrow (\ E\ ) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \varepsilon$

- The LL(1) parsing table:

next input token

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

leftmost non-terminal

rhs of production to use

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored

- Most programming language CFGs in their entirety are not LL(1). Having said that, many CFGs can be morphed into LL(1). Furthermore, the ideas described here can be used to build more powerful grammars needed for programming languages.