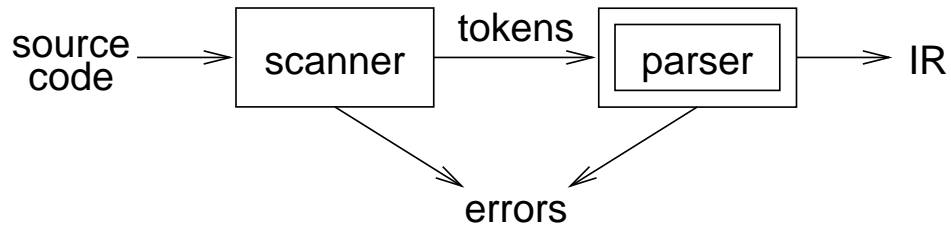


# The role of the parser

---



Parser:

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

# Syntax analysis

---

*Context free syntax* is specified with a context free grammar.

Formally, a CFG  $G$  is a 4-tuple  $(V_t, V_n, S, P)$ , where:

$V_t$  is the set of terminal symbols in the grammar.

For our purposes,  $V_t$  is the set of tokens returned by the scanner.

$V_n$  are the *nonterminals*, a set of syntactic variables that denote sets of (sub)strings occurring in the language.

These are used to impose a structure on the grammar.

$S$  is a distinguished nonterminal ( $S \in V_n$ ) denoting the entire set of strings in  $L(G)$ .

This is sometimes called a *goal symbol*.

$P$  is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language.

Each production must have a single non-terminal on its left hand side.

The set  $V = V_t \cup V_n$  is called the *vocabulary* of  $G$

## Notation and terminology

---

- $a, b, c, \dots \in V_t$
- $A, B, C, \dots \in V_n$
- $U, V, W, \dots \in V$
- $\alpha, \beta, \gamma, \dots \in V^*$
- $u, v, w, \dots \in V_t^*$

If  $A \rightarrow \gamma$  then  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  is a *single-step derivation* using  $A \rightarrow \gamma$

Similarly,  $\Rightarrow^*$  and  $\Rightarrow^+$  denote derivations of  $\geq 0$  and  $\geq 1$  steps

If  $S \Rightarrow^* \beta$  then  $\beta$  is said to be a *sentential form* of  $G$

$L(G) = \{w \in V_t^* \mid S \Rightarrow^+ w\}$ ,  $w \in L(G)$  is called a *sentence* of  $G$

Note,  $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

# Syntax analysis

---

Grammars are often written in Backus-Naur form (BNF).

Example:

```
1 | <goal> ::= <expr>
2 | <expr> ::= <expr><op><expr>
3 |           | num
4 |           | id
5 | <op> ::= +
6 | <op> ::= -
7 | <op> ::= *
8 | <op> ::= /
```

This describes simple expressions over numbers and identifiers.

In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with typewriter font or underline
3. productions as in the example

## Scanning vs. parsing

---

*Where do we draw the line?*

$$\begin{aligned}
 \textit{term} & ::= [a-zA-Z]([a-zA-Z] \mid [0-9])^* \\
 & \quad \mid 0 \mid [1-9][0-9]^* \\
 \textit{op} & ::= + \mid - \mid * \mid / \\
 \textit{expr} & ::= (\textit{term} \textit{op})^* \textit{term}
 \end{aligned}$$

Regular expressions are used to classify:

- identifiers, numbers, keywords
- REs are more concise and easier to understand for tokens than a grammar
- more efficient scanners (DFAs) can be built from REs than from arbitrary grammars

Context free grammars are used to count:

- brackets: `()`, `begin...end`, `if...then...else`
- imparting structure: expressions

Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes the compiler more manageable.

# Derivations

---

We can view the productions of a CFG as rewriting rules.

Using our example CFG:

$$\begin{aligned}
 \langle \text{goal} \rangle &\Rightarrow \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{id}, x \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{id}, x \rangle + \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle * \langle \text{expr} \rangle \\
 &\Rightarrow \langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle
 \end{aligned}$$

We have derived the sentence  $x + 2*y$ .

We denote this  $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$ .

Such a sequence of rewrites is a *derivation* or a *parse*.

The process of discovering a derivation is called *parsing*.

# Derivations

---

*At each step, we chose a non-terminal to replace.*

*This choice can lead to different derivations.*

Two are of particular interest:

*leftmost derivation*

the leftmost non-terminal is replaced at each step

*rightmost derivation*

the rightmost non-terminal is replaced at each step

*The previous example was a leftmost derivation.*

## Rightmost derivation

---

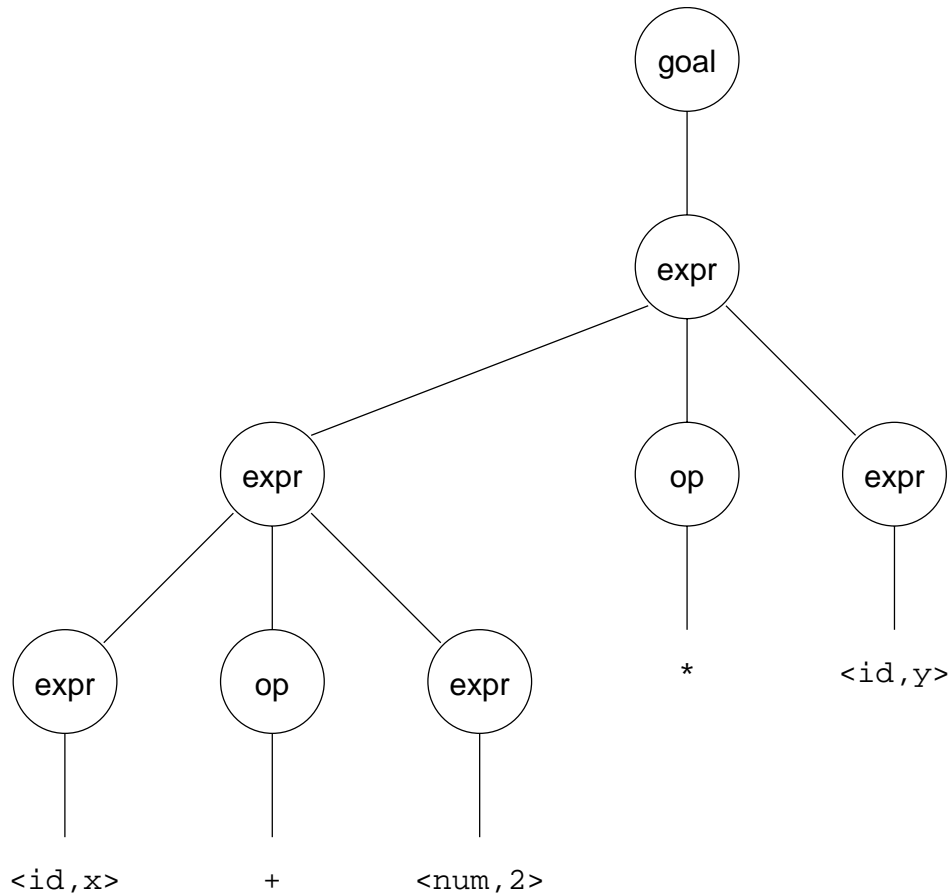
For the string  $x + 2*y$ :

```
<goal>  =>  <expr>
          =>  <expr><op><expr>
          =>  <expr><op><id,y>
          =>  <expr>*<id,y>
          =>  <expr><op><expr>*<id,y>
          =>  <expr><op><num,2>*<id,y>
          =>  <expr> + <num,2>*<id,y>
          =>  <id,x> + <num,2>*<id,y>
```

Again,  $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$ .

# Precedence

---



*Treewalk evaluation computes  $(x + 2) * y$*   
— the “wrong” answer!

Should be  $x + (2 * y)$

# Precedence

---

*These two derivations point out a problem with the grammar.*

*It has no notion of precedence, or implied order of evaluation.*

To add precedence takes additional machinery:

1	<goal>	::=	<expr>
2	<expr>	::=	<expr> + <term>
3			<expr> - <term>
4			<term>
5	<term>	::=	<term> * <factor>
6			<term> / <factor>
7			<factor>
8	<factor>	::=	num
9			id

This grammar enforces a precedence on the derivation:

- terms *must* be derived from expressions
- forces the “correct” tree

# Precedence

---

Now, for the string  $x + 2 * y$ :

```

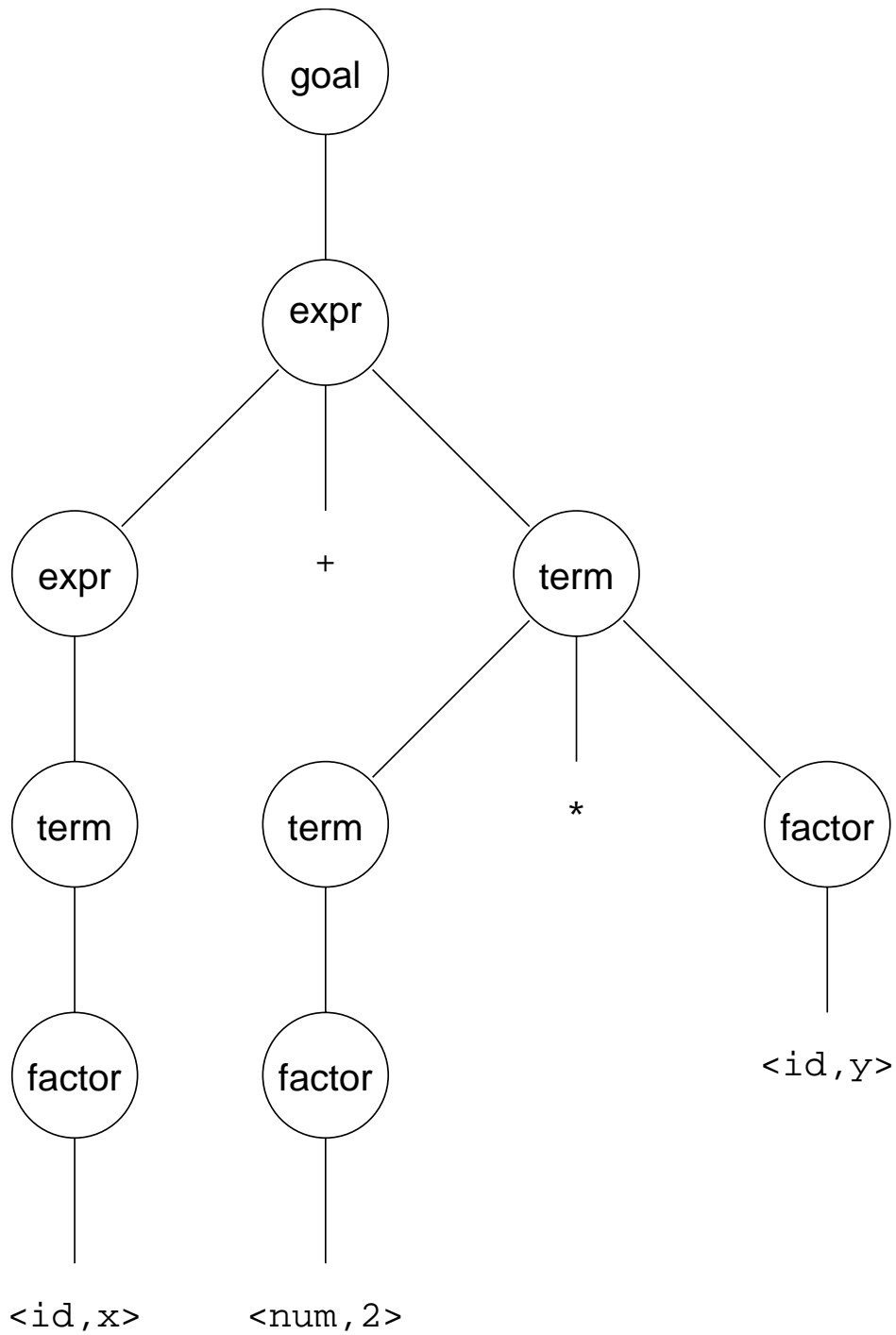
<goal>  =>  <expr>
          =>  <expr> + <term>
          =>  <expr> + <term> * <factor>
          =>  <expr> + <term> * <id,y>
          =>  <expr> + <factor> * <id,y>
          =>  <expr> + <num,2> * <id,y>
          =>  <term> + <num,2> * <id,y>
          =>  <factor> + <num,2> * <id,y>
          =>  <id,x> + <num,2> * <id,y>

```

Again,  $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$ , but this time, we build the desired tree.

# Precedence

---



*Treewalk evaluation computes  $x + (2 * y)$*

# Ambiguity

---

If a grammar has more than one derivation for a single sentential form, then it is *ambiguous*

Example:

```
<stmt> ::= if <expr> then <stmt>
          | if <expr> then <stmt> else <stmt>
          | other stmts
```

Consider deriving the sentential form:

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$$

It has two derivations.

This ambiguity is purely grammatical.

It is a *context free* ambiguity.

# Ambiguity

---

May be able to eliminate ambiguities by rearranging the grammar:

```
<stmt>      ::= <matched>
              | <unmatched>
<matched>   ::= if <expr> then <matched> else <matched>
              | other stmts
<unmatched> ::= if <expr> then <stmt>
              | if <expr> then <matched> else <unmatched>
```

This generates the same language as the ambiguous grammar, but applies the common sense rule:

*match each else with the closest unmatched then*

This is most likely the language designer's intent.

# Ambiguity

---

*Ambiguity* is often due to confusion in the context free specification.

Context sensitive confusions can arise from *overloading*.

Example:

$$a = f(17)$$

In many Algol-like languages,  $f$  could be a function or a subscripted variable.

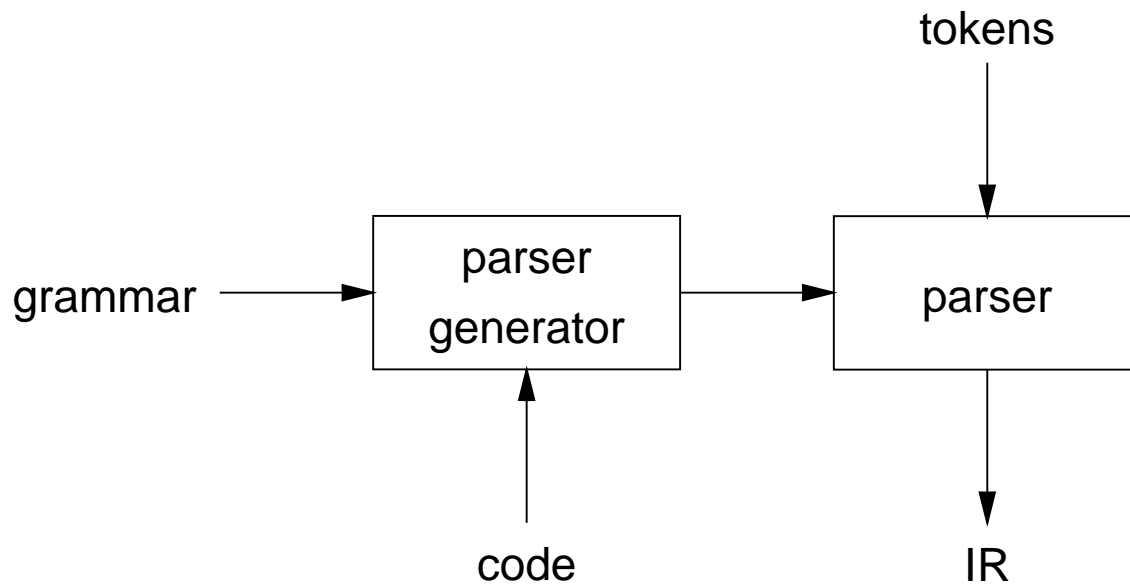
Disambiguating this statement requires context:

- need *values* of declarations
- not *context free*
- really an issue of *type*

*Rather than complicate parsing, we will handle this separately.*

# Parsing: the big picture

---



*Our goal is a flexible parser generator system*

# Top-down versus bottom-up

---

## *Top-down parsers*

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

## *Bottom-up parsers*

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (*recognize valid prefixes*)
- use a stack to store both state and sentential forms

# Top-down parsing

---

*A top-down parser starts with the root of the parse tree, labelled with the start or goal symbol of the grammar.*

To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labelled  $A$ , select a production  $A \rightarrow \alpha$  and construct the appropriate child for each symbol of  $\alpha$
2. When a terminal is added to the fringe that doesn't match the input string, backtrack
3. Find the next node to be expanded (must have a label in  $V_n$ )

The key is selecting the right production in step 1

$\Rightarrow$  should be guided by the input string

# Simple expression grammar

---

Recall our grammar for simple expressions:

1		<goal>	::=	<expr>
2		<expr>	::=	<expr> + <term>
3				<expr> - <term>
4				<term>
5		<term>	::=	<term> * <factor>
6				<term> / <factor>
7				<factor>
8		<factor>	::=	num
9				id

Consider the input string  $x - 2 * y$

# Example

Prod'n	Sentential form	Input
-	<goal>	$\uparrow x - 2 * y$
1	<expr>	$\uparrow x - 2 * y$
2	<expr> + <term>	$\uparrow x - 2 * y$
4	<term> + <term>	$\uparrow x - 2 * y$
7	<factor> + <term>	$\uparrow x - 2 * y$
9	id + <term>	$\uparrow x - 2 * y$
-	id + <term>	$x \uparrow - 2 * y$
-	<expr>	$\uparrow x - 2 * y$
3	<expr> - <term>	$\uparrow x - 2 * y$
4	<term> - <term>	$\uparrow x - 2 * y$
7	<factor> - <term>	$\uparrow x - 2 * y$
9	id - <term>	$\uparrow x - 2 * y$
-	id - <term>	$x \uparrow - 2 * y$
-	id - <term>	$x - \uparrow 2 * y$
7	id - <factor>	$x - \uparrow 2 * y$
8	id - num	$x - \uparrow 2 * y$
-	id - num	$x - 2 \uparrow * y$
-	id - <term>	$x - \uparrow 2 * y$
5	id - <term> * <factor>	$x - \uparrow 2 * y$
7	id - <factor> * <factor>	$x - \uparrow 2 * y$
8	id - num * <factor>	$x - \uparrow 2 * y$
-	id - num * <factor>	$x - 2 \uparrow * y$
-	id - num * <factor>	$x - 2 * \uparrow y$
9	id - num * id	$x - 2 * \uparrow y$
-	id - num * id	$x - 2 * y \uparrow$

## Example

---

Another possible parse for  $x - 2 * y$

Prod'n	Sentential form	Input
–	<goal>	$\uparrow x - 2 * y$
1	<expr>	$\uparrow x - 2 * y$
2	<expr> + <term>	$\uparrow x - 2 * y$
2	<expr> + <term> + <term>	$\uparrow x - 2 * y$
2	<expr> + <term> + ...	$\uparrow x - 2 * y$
2	<expr> + <term> + ...	$\uparrow x - 2 * y$
2	...	$\uparrow x - 2 * y$

If the parser makes wrong choices, expansion doesn't terminate.

This isn't a good property for a parser to have.

(Parsers should terminate!)

# Left-recursion

---

*Top-down parsers cannot handle left-recursion in a grammar*

Formally, a grammar is *left-recursive* if

$\exists A \in V_n$  such that  $A \Rightarrow^+ A\alpha$  for some string  $\alpha$

*Our simple expression grammar is left-recursive*

## Eliminating left-recursion

---

*To remove left-recursion, we can transform the grammar*

Consider the grammar fragment:

$$\begin{array}{l} A ::= A\alpha \\ \quad | \beta \end{array}$$

where  $\alpha$  and  $\beta$  do not start with  $A$

We can rewrite this as:

$$\begin{array}{l} A ::= \beta A' \\ A' ::= \alpha A' \\ \quad | \epsilon \end{array}$$

where  $A'$  is a new non-terminal

*This fragment contains no left-recursion*

## Example

---

Our expression grammar contains two cases of left-recursion

$$\begin{array}{lcl}
 \langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle + \langle \text{term} \rangle \\
 & & | \langle \text{expr} \rangle - \langle \text{term} \rangle \\
 & & | \langle \text{term} \rangle \\
 \langle \text{term} \rangle & ::= & \langle \text{term} \rangle * \langle \text{factor} \rangle \\
 & & | \langle \text{term} \rangle / \langle \text{factor} \rangle \\
 & & | \langle \text{factor} \rangle
 \end{array}$$

Applying the transformation gives

$$\begin{array}{lcl}
 \langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
 \langle \text{expr}' \rangle & ::= & + \langle \text{term} \rangle \langle \text{expr}' \rangle \\
 & & | - \langle \text{term} \rangle \langle \text{expr}' \rangle \\
 & & | \epsilon \\
 \langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
 \langle \text{term}' \rangle & ::= & * \langle \text{factor} \rangle \langle \text{term}' \rangle \\
 & & | / \langle \text{factor} \rangle \langle \text{term}' \rangle \\
 & & | \epsilon
 \end{array}$$

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

## Example

---

This cleaner grammar defines the same language

```

1 | <goal>      ::= <expr>
2 | <expr>      ::= <term> + <expr>
3 |             | <term> - <expr>
4 |             | <term>
5 | <term>      ::= <factor> * <term>
6 |             | <factor> / <term>
7 |             | <factor>
8 | <factor>    ::= num
9 |             | id

```

It is

- right-recursive
- free of  $\epsilon$  productions

*Unfortunately, it generates different associativity*

*Same syntax, different meaning*

## Example

---

The expression grammar:

1		<goal>	::=	<expr>
2		<expr>	::=	<term> <expr'>
3		<expr'>	::=	+ <term> <expr'>
4				- <term> <expr'>
5				$\epsilon$
6		<term>	::=	<factor> <term'>
7		<term'>	::=	* <factor> <term'>
8				/ <factor> <term'>
9				$\epsilon$
10		<factor>	::=	num
11				id

Recall, we factored out left-recursion

## How much lookahead is needed?

---

*We saw that top-down parsers may need to backtrack when they select the wrong production*

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are:

**LL(1):** Left to right scan, Left-most derivation, 1-token lookahead

**LR(1):** Left to right scan, Right-most derivation, 1-token lookahead

# Predictive parsing

---

*Basic idea:*

For any two productions  $A \rightarrow \alpha|\beta$ , we would like a distinct way of choosing the correct production to expand.

For some RHS  $\alpha \in G$ , define  $\text{FIRST}(\alpha)$  as the set of tokens that appear first in some string derived from  $\alpha$

That is, for some  $w \in V_t^*$ ,  $w \in \text{FIRST}(\alpha)$  iff.  $\alpha \Rightarrow^* w\gamma$ .

*Key property:*

Whenever two productions  $A \rightarrow \alpha$  and  $A \rightarrow \beta$  both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of only one symbol!

*The example grammar has this property!*

# Left factoring

---

*What if a grammar does not have this property?*

Sometimes, we can transform a grammar to have this property.

For each non-terminal  $A$  find the longest prefix  $\alpha$  common to two or more of its alternatives.

if  $\alpha \neq \epsilon$  then replace all of the  $A$  productions

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n$$

with

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 | \beta_2 | \dots | \beta_n \end{aligned}$$

where  $A'$  is a new non-terminal.

Repeat until no two alternatives for a single non-terminal have a common prefix.

## Example

---

Consider a *right-recursive* version of the expression grammar:

1	<goal>	::=	<expr>
2	<expr>	::=	<term> + <expr>
3			<term> - <expr>
4			<term>
5	<term>	::=	<factor> * <term>
6			<factor> / <term>
7			<factor>
8	<factor>	::=	num
9			id

To choose between productions 2, 3, & 4, the parser must see past the num or id and look at the +, -, \* or /.

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \emptyset$$

This grammar *fails* the test.

Note: *This grammar is right-associative.*

## Example

---

There are two nonterminals that must be left factored:

$$\begin{array}{lcl}
 \langle \text{expr} \rangle & ::= & \langle \text{term} \rangle + \langle \text{expr} \rangle \\
 & & | \quad \langle \text{term} \rangle - \langle \text{expr} \rangle \\
 & & | \quad \langle \text{term} \rangle \\
 \langle \text{term} \rangle & ::= & \langle \text{factor} \rangle * \langle \text{term} \rangle \\
 & & | \quad \langle \text{factor} \rangle / \langle \text{term} \rangle \\
 & & | \quad \langle \text{factor} \rangle
 \end{array}$$

Applying the transformation gives us:

$$\begin{array}{lcl}
 \langle \text{expr} \rangle & ::= & \langle \text{term} \rangle \langle \text{expr}' \rangle \\
 \langle \text{expr}' \rangle & ::= & + \langle \text{expr} \rangle \\
 & & | \quad - \langle \text{expr} \rangle \\
 & & | \quad \epsilon \\
 \langle \text{term} \rangle & ::= & \langle \text{factor} \rangle \langle \text{term}' \rangle \\
 \langle \text{term}' \rangle & ::= & * \langle \text{term} \rangle \\
 & & | \quad / \langle \text{term} \rangle \\
 & & | \quad \epsilon
 \end{array}$$

## Example

---

Substituting back into the grammar yields

1		<goal>	::=	<expr>
2		<expr>	::=	<term> <expr'>
3		<expr'>	::=	+ <expr>
4				- <expr>
5				ε
6		<term>	::=	<factor> <term'>
7		<term'>	::=	* <term>
8				/ <term>
9				ε
10		<factor>	::=	num
11				id

Now, selection requires only a single token lookahead.

Note: *This grammar is still right-associative.*

# Example

Prod'n	Sentential form	Input
-	<goal>	$\uparrow x - 2 * y$
1	<expr>	$\uparrow x - 2 * y$
2	<term> <expr'>	$\uparrow x - 2 * y$
6	<factor> <term'> <expr'>	$\uparrow x - 2 * y$
11	id <term'> <expr'>	$\uparrow x - 2 * y$
-	id <term'> <expr'>	$x \uparrow - 2 * y$
9	id <expr'>	$x \uparrow - 2 * y$
4	id - <expr>	$x \uparrow - 2 * y$
-	id - <expr>	$x - \uparrow 2 * y$
2	id - <term> <expr'>	$x - \uparrow 2 * y$
6	id - <factor> <term'> <expr'>	$x - \uparrow 2 * y$
10	id - num <term'> <expr'>	$x - \uparrow 2 * y$
-	id - num <term'> <expr'>	$x - 2 \uparrow * y$
7	id - num * <term> <expr'>	$x - 2 \uparrow * y$
-	id - num * <term> <expr'>	$x - 2 * \uparrow y$
6	id - num * <factor> <term'> <expr'>	$x - 2 * \uparrow y$
11	id - num * id <term'> <expr'>	$x - 2 * \uparrow y$
-	id - num * id <term'> <expr'>	$x - 2 * y \uparrow$
9	id - num * id <expr'>	$x - 2 * y \uparrow$
5	id - num * id	$x - 2 * y \uparrow$

The next symbol determined each choice correctly.

## Back to left-recursion elimination

---

Given a left-factored CFG, to eliminate left-recursion:

if  $\exists A \rightarrow A\alpha$  then replace all of the  $A$  productions

$$A \rightarrow A\alpha|\beta|\dots|\gamma$$

with

$$A \rightarrow NA'$$

$$N \rightarrow \beta|\dots|\gamma$$

$$A' \rightarrow \alpha A'|\epsilon$$

where  $N$  and  $A'$  are new productions.

Repeat until there are no left-recursive productions.

# Generality

---

Question:

By *left factoring* and *eliminating left-recursion*, can we transform an arbitrary context free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

Given a context free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many *context free languages* do not have such a grammar:

$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$

Must look past an arbitrary number of *a*'s to discover the 0 or the 1 and so determine the derivation.

# Recursive descent parsing

---

Now, we can produce a simple recursive descent parser from the (right-associative) grammar.

```
void goal()
{
    token = next_token();
    expr();
    if (token != EOF) then
        error();
}

void expr()
{
    term();
    expr_prime();
}

void expr_prime()
{
    if ((token == PLUS) || (token == MINUS)) then
    {
        token = next_token();
        expr();
    }
}
```

# Recursive descent parsing

---

```
void term()
{
    factor();
    term_prime();
}

void term_prime()
{
    if ((token == MULT) || (token == DIV)) then
    {
        token = next_token();
        term();
    }
}

void factor()
{
    if ((token == NUM) || (token == ID)) then
        token = next_token();
    else
        error();
}
```

## Building the tree

---

*One of the key jobs of the parser is to build an intermediate representation of the source code.*

To build an abstract syntax tree, we can simply insert code at the appropriate points:

- `factor()` can stack nodes `id`, `num`
- `term_prime()` can stack nodes `*`, `/`
- `term()` can pop 3, build and push subtree
- `expr_prime()` can stack nodes `+`, `-`
- `expr()` can pop 3, build and push subtree
- `goal()` can pop and return tree

# Non-recursive predictive parsing

---

Observation:

*Our recursive descent parser encodes state information in its run-time stack, or call stack.*

Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

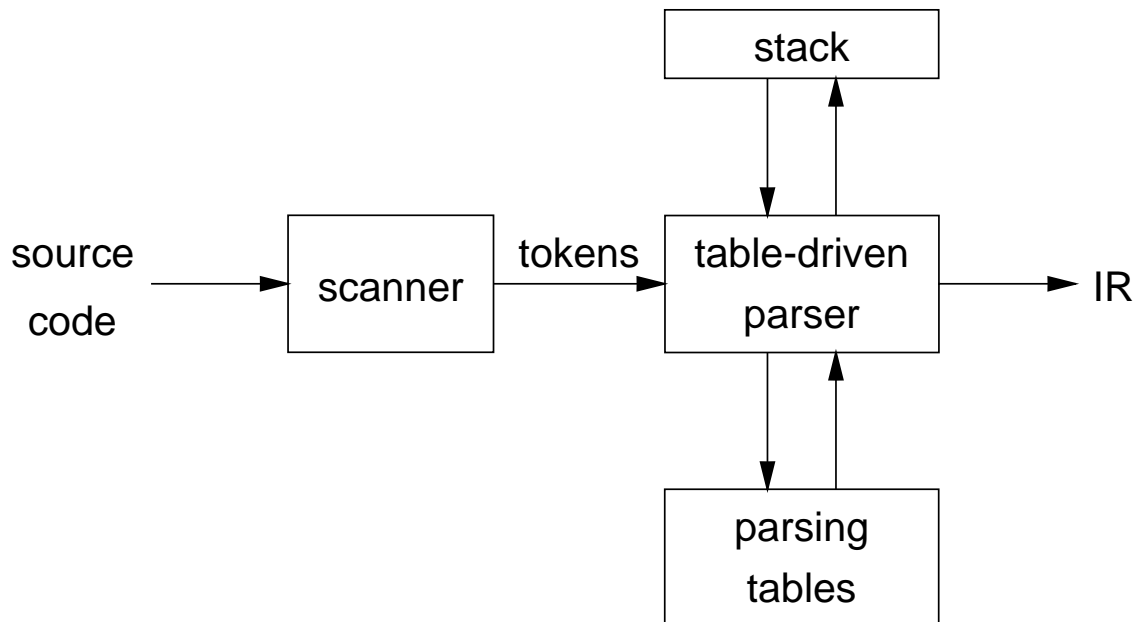
This suggests other implementation methods:

- explicit stack, hand-coded parser
- stack-based, table-driven parser

# Non-recursive predictive parsing

---

Now, a predictive parser looks like:



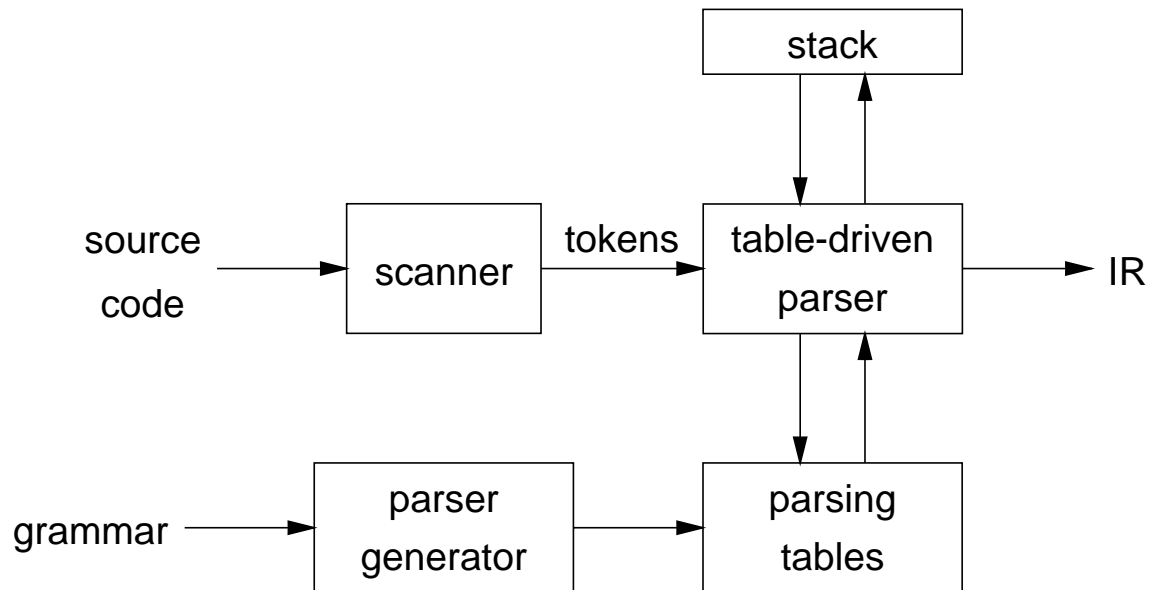
Rather than writing code, we build tables.

Building tables can be automated!

# Table-driven parsers

---

A parser generator system often looks like:



This is true for both top-down and bottom-up parsers:

- LL(1)
- LR(1)

## Non-recursive predictive parsing

---

*Input:* a string  $w$  and a parsing table  $M$  for  $G$

```
tos = 0
Stack[tos] = EOF
Stack[++tos] = Start Symbol
token = next_token()
repeat
    X = Stack[tos]
    if X is a terminal or EOF then
        if X == token then
            pop X
            token = next_token()
        else error()
    else /* X is a non-terminal */
        if  $M[X,token] == X \rightarrow Y_1Y_2\dots Y_k$  then
            pop X
            push  $Y_k, Y_{k-1}, \dots, Y_1$ 
        else error()
until X == EOF
```

# Non-recursive predictive parsing

---

*What we need now is a parsing table  $M$ .*

The expression grammar:

1	$\langle \text{goal} \rangle$	$::=$	$\langle \text{expr} \rangle$
2	$\langle \text{expr} \rangle$	$::=$	$\langle \text{term} \rangle \langle \text{expr}' \rangle$
3	$\langle \text{expr}' \rangle$	$::=$	$+ \langle \text{expr} \rangle$
4			$- \langle \text{expr} \rangle$
5			$\epsilon$
6	$\langle \text{term} \rangle$	$::=$	$\langle \text{factor} \rangle \langle \text{term}' \rangle$
7	$\langle \text{term}' \rangle$	$::=$	$* \langle \text{term} \rangle$
8			$/ \langle \text{term} \rangle$
9			$\epsilon$
10	$\langle \text{factor} \rangle$	$::=$	num
11			id

Its parse table:

	id	num	+	-	*	/	\$
$\langle \text{goal} \rangle$	1	1	-	-	-	-	-
$\langle \text{expr} \rangle$	2	2	-	-	-	-	-
$\langle \text{expr}' \rangle$	-	-	3	4	-	-	5
$\langle \text{term} \rangle$	6	6	-	-	-	-	-
$\langle \text{term}' \rangle$	-	-	9	9	7	8	9
$\langle \text{factor} \rangle$	11	10	-	-	-	-	-

# FIRST

---

For a string of grammar symbols  $\alpha$ , we define  $\text{FIRST}(\alpha)$  as:

- the set of terminal symbols that begin strings derived from  $\alpha$ :  $\{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$
- If  $\alpha \Rightarrow^* \epsilon$  then  $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$  contains the set of tokens valid in the initial position in  $\alpha$

To build  $\text{FIRST}(X)$ :

1. If  $X \in V_t$  then  $\text{FIRST}(X)$  is  $\{X\}$
2. If  $X \rightarrow \epsilon$  then add  $\epsilon$  to  $\text{FIRST}(X)$ .
3. If  $X \rightarrow Y_1Y_2\dots Y_k$ :
  - (a) Put  $\text{FIRST}(Y_1) - \{\epsilon\}$  in  $\text{FIRST}(X)$
  - (b)  $\forall i : 1 < i \leq k$ , if  $\epsilon \in \text{FIRST}(Y_1) \cap \dots \cap \text{FIRST}(Y_{i-1})$   
(i.e.,  $Y_1\dots Y_{i-1} \Rightarrow^* \epsilon$ )  
then put  $\text{FIRST}(Y_i) - \{\epsilon\}$  in  $\text{FIRST}(X)$
  - (c) If  $\epsilon \in \text{FIRST}(Y_1) \cap \dots \cap \text{FIRST}(Y_k)$  then put  $\epsilon$  in  $\text{FIRST}(X)$

Repeat until no more additions can be made.

# FOLLOW

---

For a non-terminal  $B$ , we define  $\text{FOLLOW}(B)$  as:

the set of terminals that can appear immediately to the right of  $B$  in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.

A terminal symbol has no FOLLOW set.

To build  $\text{FOLLOW}(B)$ :

1. Put  $\$$  in  $\text{FOLLOW}(\langle \text{goal} \rangle)$
2. If  $A \rightarrow \alpha B \beta$ :
  - (a) Put  $\text{FIRST}(\beta) - \{\epsilon\}$  in  $\text{FOLLOW}(B)$
  - (b) If  $\beta = \epsilon$  (i.e.,  $A \rightarrow \alpha B$ ) or  $\epsilon \in \text{FIRST}(\beta)$  (i.e.,  $\beta \Rightarrow^* \epsilon$ ) then put  $\text{FOLLOW}(A)$  in  $\text{FOLLOW}(B)$

Repeat until no more additions can be made

# LOOKAHEAD

---

For a production rule  $A \rightarrow \alpha$ , we define  $\text{LOOKAHEAD}(A \rightarrow \alpha)$  as:

the set of terminals which can appear next in the input when recognising production rule  $A \rightarrow \alpha$

Thus, a production rule's  $\text{LOOKAHEAD}$  set specifies the tokens which should appear next in the input before the production rule is applied.

To build  $\text{LOOKAHEAD}(A \rightarrow \alpha)$ :

1. Put  $\text{FIRST}(\alpha) - \{\epsilon\}$  in  $\text{LOOKAHEAD}(A \rightarrow \alpha)$
2. If  $\epsilon \in \text{FIRST}(\alpha)$   
then put  $\text{FOLLOW}(A)$  in  $\text{LOOKAHEAD}(A \rightarrow \alpha)$

A grammar  $G$  is  $\text{LL}(1)$  iff. for each set of productions  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ :

$\text{LOOKAHEAD}(A \rightarrow \alpha_1), \text{LOOKAHEAD}(A \rightarrow \alpha_2), \dots, \text{LOOKAHEAD}(A \rightarrow \alpha_n)$  are all pairwise disjoint

# LL(1) grammars

---

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A  $\epsilon$ -free grammar where each alternative expansion for  $A$  begins with a distinct terminal is a *simple* LL(1) grammar.

Example

$$S \rightarrow aS|a$$

is not LL(1) because  $\text{LOOKAHEAD}(S \rightarrow aS) = \text{LOOKAHEAD}(S \rightarrow a) = \{a\}$

$$\begin{aligned} S &\rightarrow aS' \\ S' &\rightarrow aS'|\epsilon \end{aligned}$$

accepts the same language and is LL(1)

# LL(1) parse table construction

---

*Input:* Grammar  $G$

*Output:* Parsing table  $M$

*Method:*

1.  $\forall$  productions  $A \rightarrow \alpha$ :  
 $\forall a \in \text{LOOKAHEAD}(A \rightarrow \alpha)$ , add  $A \rightarrow \alpha$  to  $M[A, a]$
2. Set each undefined entry of  $M$  to error

If  $\exists M[A, a]$  with multiple entries then grammar is not LL(1).

# Example

The expression grammar:

$$\begin{array}{l|l}
 S \rightarrow E & T \rightarrow FT' \\
 E \rightarrow TE' & T' \rightarrow *T \mid /T \mid \epsilon \\
 E' \rightarrow +E \mid -E \mid \epsilon & F \rightarrow \text{id} \mid \text{num}
 \end{array}$$

	FIRST	FOLLOW
$S$	{num,id}	{ $\$$ }
$E$	{num,id}	{ $\$$ }
$E'$	{ $\epsilon$ , +, -}	{ $\$$ }
$T$	{num,id}	{+, -, $\$$ }
$T'$	{ $\epsilon$ , *, /}	{+, -, $\$$ }
$F$	{num,id}	{+, -, *, /, $\$$ }
id	{id}	-
num	{num}	-
*	{*}	-
/	{/}	-
+	{+}	-
-	{-}	-

	LOOKAHEAD
$S \rightarrow E$	{num,id}
$E \rightarrow TE'$	{num,id}
$E' \rightarrow +E$	{+}
$E' \rightarrow -E$	{-}
$E' \rightarrow \epsilon$	{ $\$$ }
$T \rightarrow FT'$	{num,id}
$T' \rightarrow *T$	{*}
$T' \rightarrow /T$	{/}
$T' \rightarrow \epsilon$	{+, -, $\$$ }
$F \rightarrow \text{id}$	{id}
$F \rightarrow \text{num}$	{num}

	id	num	+	-	*	/	$\$$
$S$	$S \rightarrow E$	$S \rightarrow E$	-	-	-	-	-
$E$	$E \rightarrow TE'$	$E \rightarrow TE'$	-	-	-	-	-
$E'$	-	-	$E' \rightarrow +E$	$E' \rightarrow -E$	-	-	$E' \rightarrow \epsilon$
$T$	$T \rightarrow FT'$	$T \rightarrow FT'$	-	-	-	-	-
$T'$	-	-	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	$T' \rightarrow *T$	$T' \rightarrow /T$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow \text{id}$	$F \rightarrow \text{num}$	-	-	-	-	-

## Building the tree

---

Again, we insert code at the right points.

```

tos = 0
Stack[tos] = EOF
Stack[++tos] = root node
Stack[++tos] = Start Symbol
token = next_token()
repeat
    X = Stack[tos]
    if X is a terminal or EOF then
        if X == token then
            pop X
            token = next_token()
            pop and fill in node
        else error()
    else /* X is a non-terminal */
        if  $M[X, token] == X \rightarrow Y_1 Y_2 \dots Y_k$  then
            pop X
            pop node for X
            build node for each child and
            make it a child of node for X
            push  $\{n_k, Y_k, n_{k-1}, Y_{k-1}, \dots, n_1, Y_1\}$ 
        else error()
until X == EOF

```

# A grammar that is not LL(1)

---


$$\begin{aligned} \langle \text{stmt} \rangle & ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \\ & \quad | \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\ & \quad | \dots \end{aligned}$$

Left-factored:

$$\begin{aligned} \langle \text{stmt} \rangle & ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \langle \text{stmt}' \rangle \mid \dots \\ \langle \text{stmt}' \rangle & ::= \text{else } \langle \text{stmt} \rangle \mid \epsilon \end{aligned}$$

$$\text{FIRST}(\langle \text{stmt}' \rangle) = \{\epsilon, \text{else}\}$$

$$\text{FOLLOW}(\langle \text{stmt}' \rangle) = \{\text{else}, \$\}$$

$$\text{LOOKAHEAD}(\langle \text{stmt}' \rangle ::= \text{else } \langle \text{stmt} \rangle) = \{\text{else}\}$$

$$\text{LOOKAHEAD}(\langle \text{stmt}' \rangle ::= \epsilon) = \{\text{else}, \$\}$$

On seeing else, conflict between choosing  
 $\langle \text{stmt}' \rangle ::= \text{else } \langle \text{stmt} \rangle$  and  $\langle \text{stmt}' \rangle ::= \epsilon$

$\Rightarrow$  grammar is not LL(1)!

The fix:

Put priority on  $\langle \text{stmt}' \rangle ::= \text{else } \langle \text{stmt} \rangle$  to  
 associate else with closest previous then.

# Error recovery

---

Key notion:

- For each non-terminal, construct a set of terminals on which the parser can synchronize
- When an error occurs looking for  $A$ , scan until an element of  $\text{SYNCH}(A)$  is found

Building SYNCH:

1.  $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNCH}(A)$
2. place keywords that start statements in  $\text{SYNCH}(A)$
3. add symbols in  $\text{FIRST}(A)$  to  $\text{SYNCH}(A)$

If we can't match a terminal on top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

(i.e.,  $\text{SYNCH}(a) = V_t - \{a\}$ )