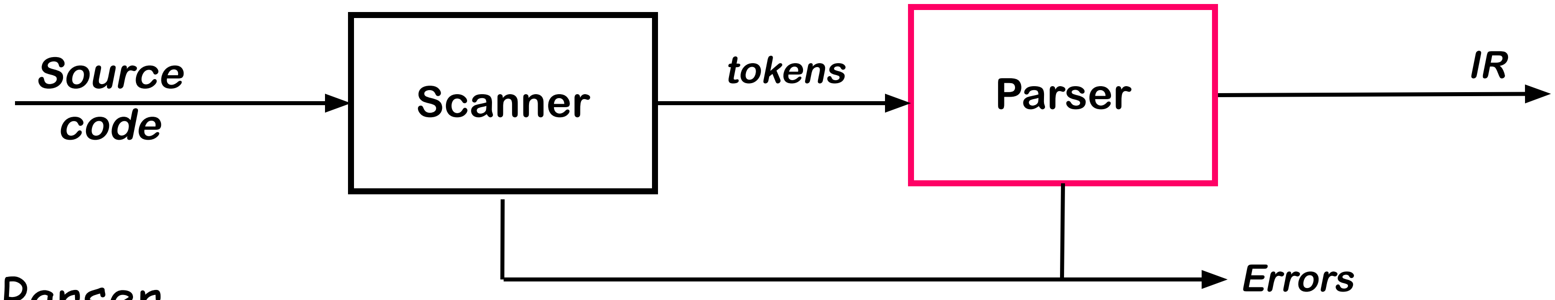
CENG513 Compiler Design and Construction
Parsing

Note by Işıl ÖZ:

Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

The Front End



Parser

- Checks the stream of <u>words</u> and their <u>parts of speech</u> (produced by the scanner) for grammatical correctness
- Determines if the input is <u>syntactically</u> well formed

- Need a mathematical model of syntax — a grammar *G*
- Need an algorithm for testing membership in *L(G)*
- Builds an IR representation of the code

The Study of Parsing

The process of discovering a *derivation* for some sentence

- If some string of words <u>*s*</u> is in the language defined by <u>*G*</u> we say that *G* <u>*derives*</u> *s*
- For a stream of words *s* and a grammar *G*, the parser tries to build a constructive proof that *s can be derived in G* → parsing
- Based on a mathematical model and an algorithm
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Parsing Algorithms

Top-down parsing

- Match the input stream against the productions of the grammar by predicting the next word
- Generated LL(1) parsers & hand-coded recursive descent parsers

Bottom-up parsing (will be skipped)

- Work from low-level detail—the actual sequence of words—and accumulate context until the derivation is apparent
- Generated LR(1) parsers

Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — a regular expression for arithmetic expressions

$$\begin{aligned} \textit{Term} &\rightarrow [\text{a-zA-Z}]\,([\text{a-zA-Z}] \mid [\underline{0}\text{-}\underline{9}])^* \\ \textit{Op} &\rightarrow \underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/} \\ \textit{Expr} &\rightarrow (\,\textit{Term}\ \textit{Op}\,)^*\ \textit{Term} \end{aligned}$$

$$([\text{a-zA-Z}]\,([\text{a-zA-Z}] \mid [\underline{0}\text{-}\underline{9}])^*\ (\underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/}))^*\ [\text{a-zA-Z}]\,([\text{a-zA-Z}] \mid [\underline{0}\text{-}\underline{9}])$$

Of course, this would generate a DFA …

If REs are so useful … Why not use them for everything?

⇒ Cannot add parenthesis, brackets, begin-end pairs, …

Why Not Use Regular Languages & DFAs?

Not all languages are regular (RL's $\subset$ CFL's $\subset$ CSL's)

You cannot construct DFA's to recognize these languages (DFAs cannot count)

- $L = \{ a^n b^n \mid n>=0\}$
- $L = \{ a^n b^m \mid n>=0, m>n\}$

Neither of these is a regular language *(nor an RE)*

Specifying Syntax with a Grammar

Syntax is specified by a *grammar*: a collection of rules that define, mathematically, which strings of symbols are valid sentences

A class of grammars called **context-free grammars** provides this power

Formally, a context-free grammar is a four tuple, $G = (S,N,T,P)$

- S is the *start/goal symbol* *(set of strings in L(G))*
- N is a set of *nonterminal symbols* *(syntactic variables)*
- T is a set of *terminal symbols* *(words)*
- P is a set of *productions* or *rewrite rules* $(P : N \rightarrow (N \cup T)^{+})$

Context-free Grammars

What makes a grammar "context free"?

The *CFG* that defines the set of noises sheep normally make, SheepNoise grammar:

$$\begin{aligned} SheepNoise &\rightarrow \underline{\text{baa}}\ SheepNoise \\ &\mid\ \underline{\text{baa}} \end{aligned}$$

Productions have a single nonterminal on the left hand side. Production rules can be applied to a nonterminal symbol regardless of its context.

⇒ The grammar is context-free.

A context-sensitive grammar can have ≥ 1 nonterminal on lhs.

Notice that *L(SheepNoise)* is actually a regular language: $\underline{\text{baa}}^{+}$

Backus-Naur Form (BNF)

```
⟨SheepNoise⟩  ::=  baa ⟨SheepNoise⟩
                |  baa
```

Sample Derivations in Tabular Form

1. *SheepNoise* → baa *SheepNoise*
2. *SheepNoise* → baa

S= {SheepNoise}

N= {SheepNoise}

T= {baa}

P= { *SheepNoise* → baa *SheepNoise*
SheepNoise → baa}

Rule	Sentential Form
	SheepNoise
2	baa

Rule	Sentential Form
	SheepNoise
1	baa *SheepNoise*
2	baa baa

Rule	Sentential Form
	SheepNoise
1	baa *SheepNoise*
1	baa baa *SheepNoise*
	. . . and so on . . .
1	baa . . . baa *SheepNoise*
2	baa baa . . . baa

A Useful Grammar

To explore the uses of CFGs, we need a grammar

0	*Expr*	$\rightarrow$	*Expr Op Expr*
1		|	<u>number</u>
2		|	<u>id</u>
3	*Op*	$\rightarrow$	+
4		|	-
5		|	*
6		|	/

S= {Expr}

N= {Expr, Op}

T= {<u>number</u>, <u>id</u>, +, -, *, /}

P= { *Expr*$\rightarrow$*Expr Op Expr*
Expr$\rightarrow$<u>number</u>
Expr$\rightarrow$<u>id</u>
Op $\rightarrow$+
Op $\rightarrow$-
Op $\rightarrow$*
Op $\rightarrow$/
}

A Useful Grammar

<u>x</u> – <u>2</u> * <u>y</u>

0	*Expr*	→	*Expr Op Expr*
1		|	<u>number</u>
2		|	<u>id</u>
3	*Op*	→	+
4		|	-
5		|	*
6		|	/

Rule	*Sentential Form*
—	*Expr*
0	*Expr Op Expr*
2	<id,<u>x</u>> *Op Expr*
4	<id,<u>x</u>> - *Expr*
0	<id,<u>x</u>> - *Expr Op Expr*
1	<id,<u>x</u>> - <num,<u>2</u>> *Op Expr*
5	<id,<u>x</u>> - <num,<u>2</u>> * *Expr*
2	<id,<u>x</u>> - <num,<u>2</u>> * <id,<u>y</u>>

- Such a sequence of rewrites is called a *derivation*
- Process of discovering a derivation is called *parsing*

We denote this derivation: *Expr* $\Rightarrow^*$ <u>id</u> – <u>num</u> * <u>id</u>

Derivations

The point of parsing is to construct a derivation

- At each step, we choose a nonterminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- *Leftmost derivation* — replace leftmost NT at each step
- *Rightmost derivation* — replace rightmost NT at each step

These are the two *systematic* derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a *leftmost* derivation

The Two Derivations for *x* – *2* * *y*

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

Leftmost derivation

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

Rightmost derivation

In both cases, *Expr* ⇒* id – num * id

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

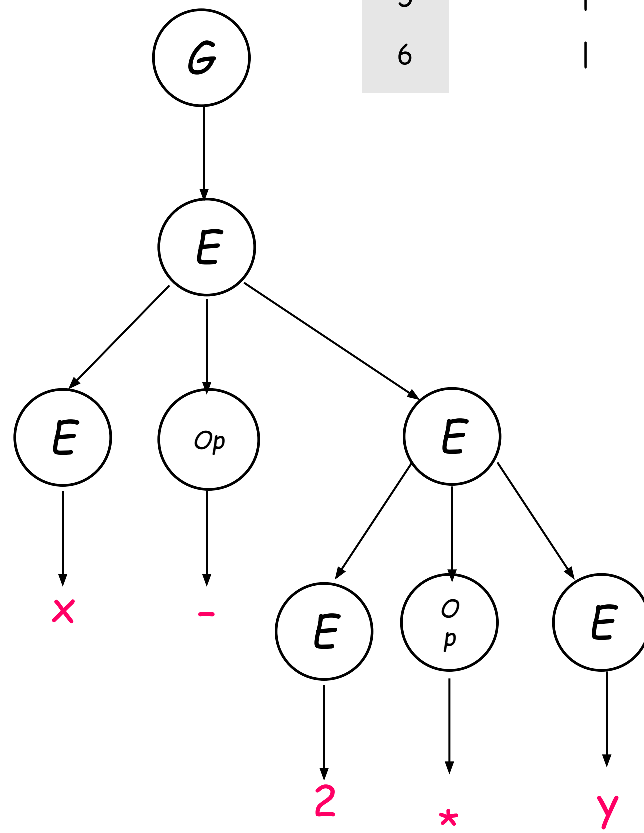
0	*Expr*	→	*Expr Op Expr*
1		|	number
2		|	id
3	*Op*	→	+
4		|	-
5		|	*
6		|	/

0	*Expr*	→	*Expr Op Expr*
1		|	<u>number</u>
2		|	<u>id</u>
3	*Op*	→	+
4		|	-
5		|	*
6		|	/

Derivations and Parse Trees

Leftmost derivation

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



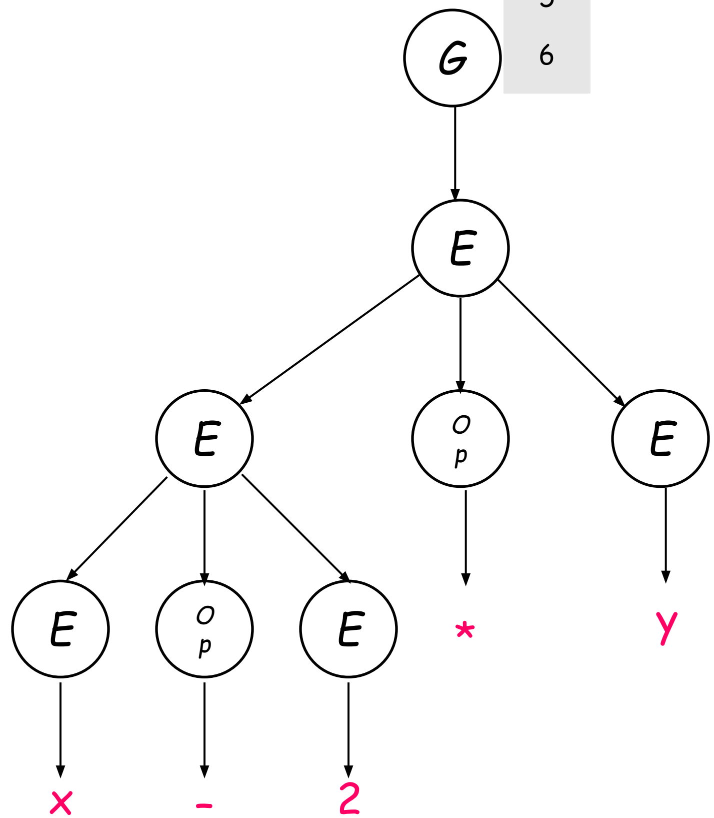
This evaluates as <u>x</u> - (<u>2</u> * <u>y</u>)

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

Derivations and Parse Trees

Rightmost derivation

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



This evaluates as (x – 2) * y

Derivations and Precedence

These two derivations point out a problem with the grammar:
It has no notion of <u>*precedence*</u>*, or implied order of evaluation*

To add precedence

- Create a nonterminal for each *level of precedence*
- Isolate the corresponding part of the grammar
- Force the parser to recognize <u>high precedence</u> subexpressions first

For algebraic expressions

- Parentheses first (*level 1*)
- Multiplication and division, next (*level 2*)
- Subtraction and addition, last (*level 3*)

Derivations and Precedence

Adding the standard algebraic precedence produces:

	0	*Goal*	→	*Expr*
level 3	1	*Expr*	→	*Expr* + *Term*
	2		|	*Expr* - *Term*
	3		|	*Term*
level 2	4	*Term*	→	*Term* * *Factor*
	5		|	*Term* / *Factor*
	6		|	*Factor*
level 1	7	*Factor*	→	(*Expr*)
	8		|	number
	9		|	id

This grammar is slightly larger

- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces the same parse tree under leftmost & rightmost derivations
- Correctness trumps the speed of the parser

*Let's see how it parses x - 2 * y*

Cannot handle precedence in an RE for expressions

Introduced parentheses, too (beyond power of an RE)

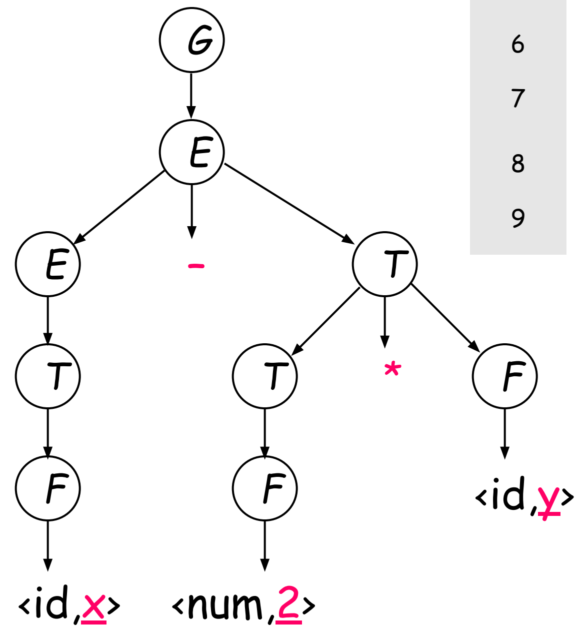
One form of the "classic expression grammar"

Derivations and Precedence

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	*(Expr)*
8		|	number
9		|	id

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
2	*Expr - Term*
4	*Expr - Term * Factor*
9	*Expr - Term* * <id,y>
6	*Expr - Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
3	*Term* - <num,2> * <id,y>
6	*Factor* - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>

The rightmost derivation



Its parse tree

This evaluates as x – (2 * y), along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same expression, because the grammar directly and explicitly encodes the desired precedence.

Ambiguous Grammars

Let's leap back to our simple expression grammar.

It had other problems.

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

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

- This grammar allows multiple leftmost derivations for x - 2 * y
- Hard to automate derivation if > 1 choice
- The grammar is *ambiguous*

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

Two Leftmost Derivations for $x - 2 * y$

The Difference:

- Different productions chosen on the second step

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

Original choice

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

New choice

- Both derivations succeed in producing $x - 2 * y$

This evaluates as x - (2 * y)

This evaluates as (x - 2) * y

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single *sentential form*, the grammar is *ambiguous*
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is *ambiguous*
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar
  - However, they must have the same parse tree!

Classic example — the *if*-*then*-*else* problem

Stmt → <u>*if*</u> *Expr* <u>*then*</u> *Stmt*
| <u>*if*</u> *Expr* <u>*then*</u> *Stmt* <u>*else*</u> *Stmt*
| … *other stmts* …

This ambiguity is inherent in the grammar

$Stmt \rightarrow$ <u>if</u> *Expr* <u>then</u> *Stmt*

$|$ <u>if</u> *Expr* <u>then</u> *Stmt* <u>else</u>

Stmt

$|$ … *other stmts* …

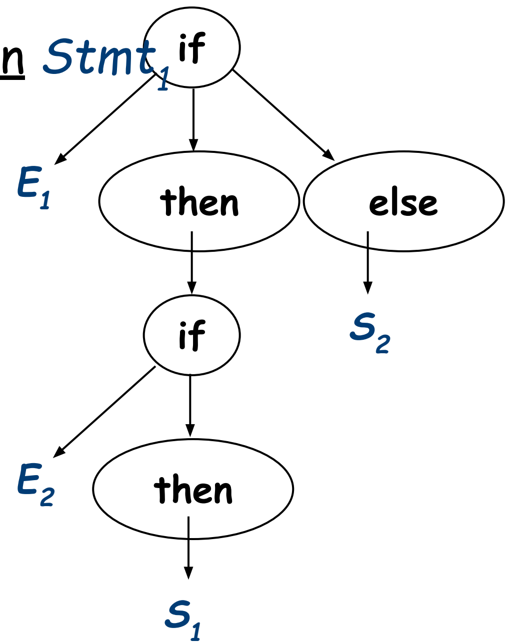
Ambiguity

This sentential form has two derivations

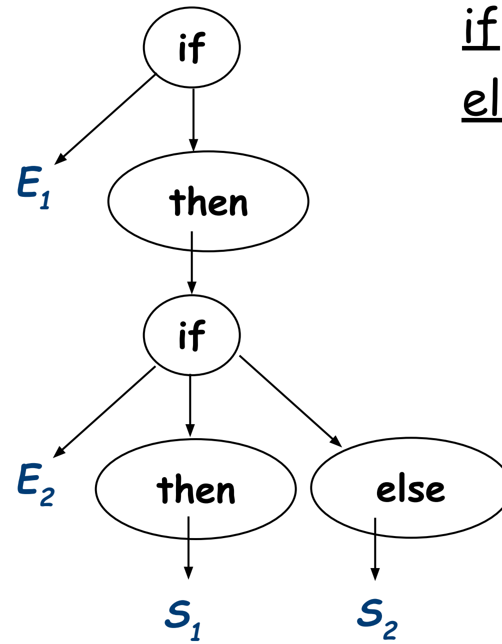
<u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$

<u>if</u> $Expr_1$ <u>then</u>
    <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$
<u>else</u> $Stmt_2$

<u>if</u> $Expr_1$ <u>then</u>
    <u>if</u> $Expr_2$ <u>then</u> Str
    <u>else</u> $Stmt_2$



production 2, then production 1



production 1, then production 2

Ambiguity

The grammar forces the structure to match the desired meaning.

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each <u>else</u> to innermost unmatched <u>if</u> *(common sense rule)*

0	*Stmt*	→	<u>if</u> *Expr* <u>then</u> *Stmt*
1		|	<u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt*
2		|	*Other Statements*
3	*WithElse*	→	<u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *WithElse*
4		|	*Other Statements*

With this grammar, example has only one rightmost derivation

Intuition: once into *WithElse*, we cannot generate an unmatched <u>else</u>

No Ambiguity

<u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$

Rule	*Sentential Form*
—	*Stmt*
0	<u>if</u> *Expr* <u>then</u> *Stmt*
1	<u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt*
2	<u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> S_2
4	<u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> S_1 <u>else</u> S_2
?	<u>if</u> *Expr* <u>then</u> <u>if</u> E_2 <u>then</u> S_1 <u>else</u> S_2
?	<u>if</u> E_1 <u>then</u> <u>if</u> E_2 <u>then</u> S_1 <u>else</u> S_2

Other productions to derive *Exprs*

This grammar has only one rightmost derivation for the example

No Ambiguity

<u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$

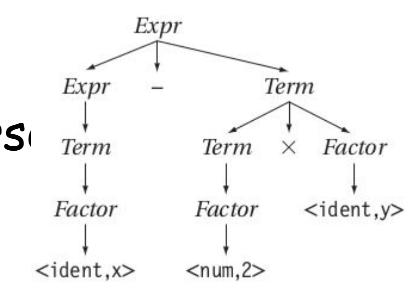
Rule	Sentential Form
—	*Stmt*
1	<u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt*
2	<u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> S_2
4	<u>if</u> *Expr* <u>then</u> S_1 <u>else</u> S_2
	No derivation possible, with rule 1

This grammar has only one rightmost derivation for the example

Parsing

- We can derive sentences that are in our language L(G) for our grammar G
- Compiler must infer a derivation for a given input string
- Parsing: Constructing a derivation from a specific input sentence

- Input: a stream o⟨ident, x⟩ - ⟨num, 2⟩ × ⟨ident, y⟩ntactic categories (returned from the scanner)
- Output
  - Derivation for the input program → Building a pars
  - Indication that the input is not a valid program



Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

A top-down parser starts with the root of the parse tree

The root node is labeled with the *goal symbol* of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until lower fringe of the parse tree matches the input string (the input stream has been exhausted)

1. At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
2. When a terminal symbol is added and it doesn't match the fringe, backtrack
3. Find the next node to be expanded *(label $\in$ NT)*

The key is picking the right production in step 1

– *That choice should be guided by the input string*

Classic Expression Grammar

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	<u>(</u> *Expr* <u>)</u>
8		|	<u>number</u>
9		|	<u>id</u>

And the input <u>x</u> – <u>2</u> * <u>y</u>

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

Example

Let's try x - 2 * y :

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y

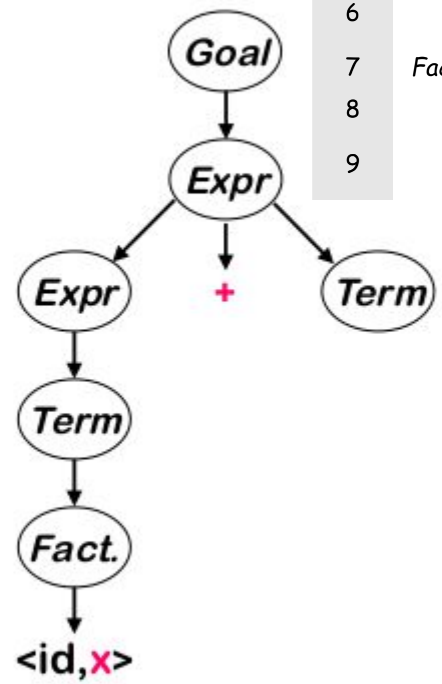
↑ is the position in the input buffer

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

Example

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr* +*Term*	↑x - 2 * y
3	*Term* +*Term*	↑x - 2 * y
6	*Factor* +*Term*	↑x - 2 * y
9	<id,x> +*Term*	↑x - 2 * y
→	<id,x> +*Term*	x ↑- 2 * y



This worked well, except that "–" doesn't match "+"

The parser must backtrack

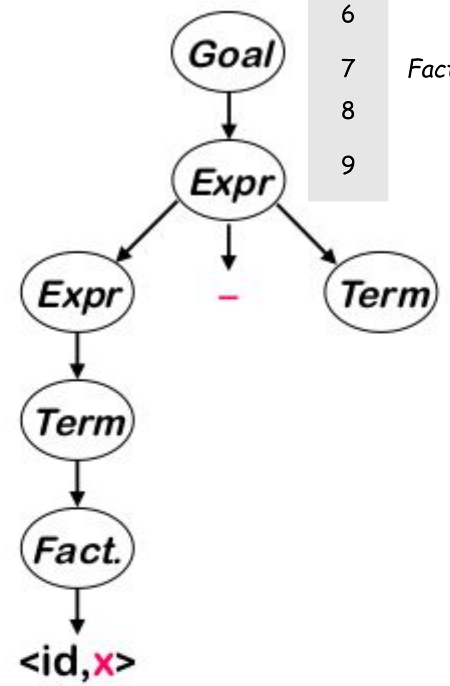
↑ *is the position in the input buffer*

Example

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	*(Expr)*
8		|	number
9		|	id

Continuing with x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
2	*Expr - Term*	↑x - 2 * y
3	*Term - Term*	↑x - 2 * y
6	*Factor - Term*	↑x - 2 * y
9	<id,x> - *Term*	↑x - 2 * y
→	<id,x> - *Term*	x ↑- 2 * y
→	<id,x> - *Term*	x - ↑2 * y



Now, "-" and "-" match

Now we can expand Term to match "2"

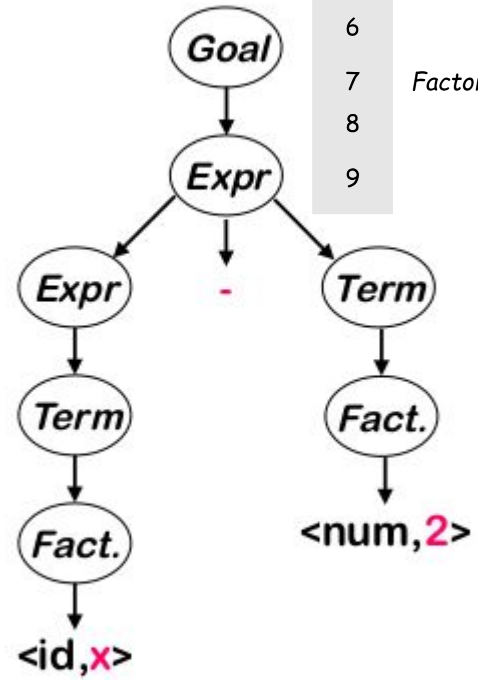
⇒ Now, we need to expand *Term* - the last *NT* on the fringe

Example

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	*(Expr)*
8		|	number
9		|	id

Trying to match the "2" in x - 2 * y :

Rule	Sentential Form	Input
→	<id,x> - *Term*	x - ↑2 * y
6	<id,x> - *Factor*	x - ↑2 * y
8	<id,x> - <num,2>	x - ↑2 * y
→	<id,x> - <num,2>	x - 2 ↑* y



Where are we?

- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

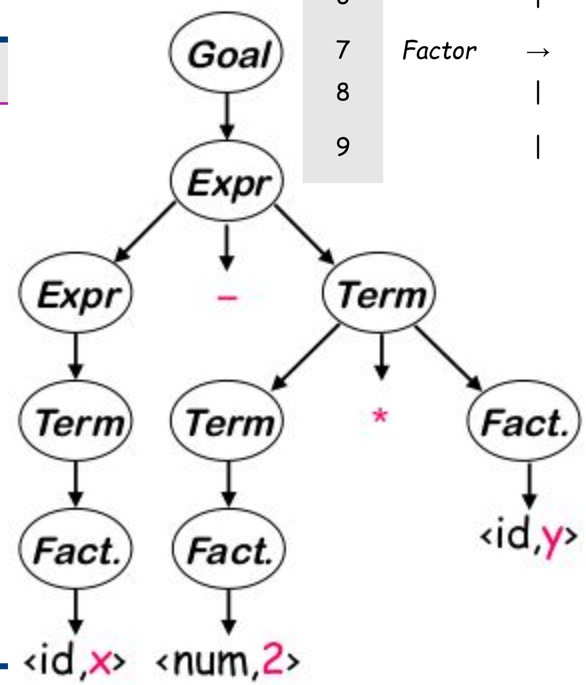
⇒ Need to backtrack

Example

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	number
9		|	id

Trying again with "2" in x - 2 * y :

Rule	*Sentential Form*	*Input*
→	<id,x> - *Term*	x - ↑2 * y
4	<id,x> - *Term* * *Factor*	x - ↑2 * y
6	<id,x> - *Factor* * *Factor*	x - ↑2 * y
8	<id,x> - <num,2> * *Factor*	x - ↑2 * y
→	<id,x> - <num,2> * *Factor*	x - 2 ↑* y
→	<id,x> - <num,2> * *Factor*	x - 2 * ↑y
9	<id,x> - <num,2> * <id,y>	x - 2 * ↑y
→	<id,x> - <num,2> * <id,y>	x - 2 * y↑



The Point:

⇒ The parser must make the right choice when it expands a NT. Wrong choices lead to wasted effort.

Another Possible Parse

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	→	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	→	*(Expr)*
8		|	number
9		|	id

Other choices for expansion are possible

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr + Term*	↑x - 2 * y
1	*Expr + Term + Term*	↑x - 2 * y
1	*Expr + Term + Term + Term*	↑x - 2 * y
1	*And so on ....*	↑x - 2 * y

Consumes no input!

This expansion doesn't terminate

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Complications in Top-Down Parsing

- Grammars with left-recursion cause termination problems
  - Eliminate left recursion
- Choosing the wrong expansion necessitates backtracking
  - Eliminate the need to backtrack

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is *left recursive* if $\exists\ A \in NT$ such that $\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup$

If the first symbol on its right-hand side is the same as the symbol on its left-hand side

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Expr + Term*
2		|	*Expr - Term*
3		|	*Term*
4	*Term*	$\rightarrow$	*Term * Factor*
5		|	*Term / Factor*
6		|	*Factor*
7	*Factor*	$\rightarrow$	*(Expr)*
8		|	number
9		|	id

Our classic expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- In a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is always a bad property in a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$X \to X \ \alpha$$
$$\quad\quad | \ \ \beta$$

where neither α nor β start with X

We can rewrite this fragment as

$$X \to \beta \ X'$$
$$X' \to \alpha \ X'$$
$$\quad\quad | \ \ \varepsilon$$

where X' is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & \mid & \mathit{Expr} - \mathit{Term} \\
 & \mid & \mathit{Term}
\end{array}
\qquad
\begin{array}{lcl}
\mathit{Term} & \rightarrow & \mathit{Term} * \mathit{Factor} \\
 & \mid & \mathit{Term} * \mathit{Factor} \\
 & \mid & \mathit{Factor}
\end{array}
$$

Applying the transformation yields

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' & \rightarrow & +\ \mathit{Term}\ \mathit{Expr}' \\
 & \mid & -\ \mathit{Term}\ \mathit{Expr}' \\
 & \mid & \varepsilon
\end{array}
\qquad
\begin{array}{lcl}
\mathit{Term} & \rightarrow & \mathit{Factor}\ \mathit{Term}' \\
\mathit{Term}' & \rightarrow & *\ \mathit{Factor}\ \mathit{Term}' \\
 & \mid & /\ \mathit{Factor}\ \mathit{Term}' \\
 & \mid & \varepsilon
\end{array}
$$

These fragments use only right recursion

Eliminating Left Recursion

Substituting them back into the grammar yields

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	<u>(</u> *Expr* <u>)</u>
10		|	<u>number</u>
11		|	<u>id</u>

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
  - ⇒ The naïve transformation yields a right recursive grammar, which changes the implicit associativity
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.

Right-Recursive Expression Grammar

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	id

Let's try x - 2 * y :

Rule	Sentential Form	Input
	Expr	↑ x - 2 × y
1	*Term Expr'*	↑ x - 2 × y
5	*Factor Term' Expr'*	↑ x - 2 × y
11	ident *Term' Expr'*	↑ x - 2 × y
→	ident *Term' Expr'*	x ↑ - 2 × y
8	ident *Expr'*	x ↑ - 2 × y
3	ident - *Term Expr'*	x ↑ - 2 × y
→	ident - *Term Expr'*	x - ↑ 2 × y
5	ident - *Factor Term' Expr'*	x - ↑ 2 × y
10	ident - num *Term' Expr'*	x - ↑ 2 × y
→	ident - num *Term' Expr'*	x - 2 ↑ × y
6	ident - num × *Factor Term' Expr'*	x - 2 ↑ × y
→	ident - num × *Factor Term' Expr'*	x - 2 × ↑ y
11	ident - num × ident *Term' Expr'*	x - 2 × ↑ y
→	ident - num × ident *Term' Expr'*	x - 2 × y ↑
8	ident - num × ident *Expr'*	x - 2 × y ↑
4	ident - num × ident	x - 2 × y ↑

⇒ Parse with no backtracking for this case

- ⇒ Parser can always make the correct choice by comparing the next word in the input stream

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

We will focus on LL(1) grammars & predictive parsing

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α and β

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string derived from α

For the terminals, + and -, their FIRST sets contain exactly one element—the symbol itself

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	*+ Term Expr'*
3		|	*- Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	** Factor Term'*
7		|	*/ Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	id

Predictive Parsing

What about ε-productions?

The parser must compare the next word against the set of symbols that can appear immediately to the right of the ε (or, equivalently, to the right of the Expr')

The set of symbols that can be derived from any symbol that follows Expr' in the rhs of some production

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), where

FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form

Define FIRST$^+$($A \rightarrow \alpha$) as

- FIRST(α) ∪ FOLLOW(A), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

FIRST$^+$($A \rightarrow \alpha$) ∩ FIRST$^+$($A \rightarrow \beta$) = ∅

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Can write a simple routine to recognize each *lhs*
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \varnothing \text{ if } i \neq j$$

```
/* find an A */
if (current_word ∈ FIRST(A→β1))
   find a β1 and return true
else if (current_word ∈ FIRST(A→β2))
   find a β2 and return true
else if (current_word ∈ FIRST(A→β3))
   find a β3 and return true
else
   report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser.

Recursive Descent Parsing - An LL(1) Parser

Recall the expression grammar, after transformation

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	id

This produces a parser with six *mutually recursive* routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term *descent* refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Proced al)

Routines from the expression parser

```
Main()
  /* Goal → Expr */
  word ← NextWord();
  if (Expr() and word = eof)
    then proceed to the next step
    else return false

Expr()
  /* Expr → Term Expr' */
  if (Term() = false)
    then return false
    else return EPrime()

EPrime()
  /* Expr' → + Term Expr'*/
  /* Expr' → - Term Expr'*/
  if (word = + or word = -) then
    word ← NextWord()
    if (Term() = false)
      then return false
      else return EPrime()
  /* Expr' → ϵ */
  return true

Term()
  /* Term → Factor Term' */
  if (Factor() = false)
    then return false
    else return TPrime()

TPrime()
  /* Term' → × Factor Term' */
  /* Term' → ÷ Factor Term' */
  if (word = × or word = ÷ ) then
    word ← NextWord()
    if (Factor() = false)
      then return false
      else return TPrime()
  /* Term' → ϵ */
  return true

Factor()
  /* Factor → ( Expr ) */
  if (word = ( ) then
    word ← NextWord()
    if (Expr() = false)
      then return false
      else if (word ≠ ) ) then
        report syntax error
        return false
  /* Factor → num */
  /* Factor → ident */
  else if (word ≠ num and
          word ≠ ident) then
    report syntax error
    return false
  word ← NextWord()
  return true
```

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	(*Expr*)
10		|	number
11		|	id

Classic Expression Grammar

0	*Goal*	$\rightarrow$	*Expr*
1	*Expr*	$\rightarrow$	*Term Expr'*
2	*Expr'*	$\rightarrow$	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	$\rightarrow$	*Factor Term'*
6	*Term'*	$\rightarrow$	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	$\rightarrow$	<u>number</u>
10		|	<u>id</u>
11		|	<u>(</u> *Expr* <u>)</u>

Symbol	FIRST	FOLLOW
<u>num</u>	<u>num</u>	Ø
<u>id</u>	<u>id</u>	Ø
+	+	Ø
-	-	Ø
*	*	Ø
/	/	Ø
<u>(</u>	<u>(</u>	Ø
<u>)</u>	<u>)</u>	Ø
<u>eof</u>	<u>eof</u>	Ø
ε	ε	Ø
Goal	<u>(</u>,<u>id</u>,<u>num</u>	eof
Expr	<u>(</u>,<u>id</u>,<u>num</u>	<u>)</u>, eof
Expr'	+, -, ε	<u>)</u>, eof
Term	<u>(</u>,<u>id</u>,<u>num</u>	+, -, <u>)</u>, eof
Term'	*, /, ε	+,-, <u>)</u>, eof
Factor	<u>(</u>,<u>id</u>,<u>num</u>	+,-,*,/,<u>)</u>,eof

FIRST$^+$($A \rightarrow \beta$) is identical to FIRST(β) except for productions 4 and 8

FIRST$^+$(*Expr'* $\rightarrow$ ε) is {ε,<u>)</u>, eof}

FIRST$^+$(*Term'* $\rightarrow$ ε) is {ε,+,-, <u>)</u>, eof}

Classic Expression Grammar

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	number
10		|	id
11		|	(*Expr*)

Prod'n	FIRST+
0	(,id,num
1	(,id,num
2	+
3	-
4	ε,), eof
5	(,id,num
6	*
7	/
8	ε,+,-,), eof
9	number
10	id
11	(

Building Top-down Parsers for LL(1) Grammars

Given an *LL(1)* grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working (*perhaps ugly*) code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” ⇒ may need to backtrack
- Some grammars are backtrack-free

Bottom-up parsers *(LR(1), operator precedence)*

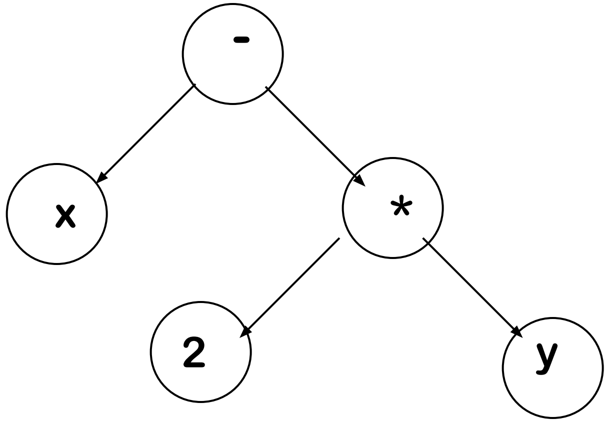
- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Summary

	Advantages	*Disadvantages*
Top-down Recursive descent, LL(1)	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

- Can use linearized form of the tree
  - Easier to manipulate than pointers

    `x 2 y * -` in postfix form

    `- * 2 y x` in prefix form

References

Chapter sections from the book:

- 3.1, 3.2, 3.3

Selected videos from compiler course from California State University:

- https://www.youtube.com/watch?v=a4H30Af55No&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=13
- https://www.youtube.com/watch?v=HXN2AGMRZWg&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=14
- https://www.youtube.com/watch?v=IAXJ3j2tB_Q&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=15
- https://www.youtube.com/watch?v=Twv3q5NNPtM&list=PL6KMWPQP_DM97Hh0PYNgJord-sANFTI3i&index=18

Kaleidoscope Parser

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html