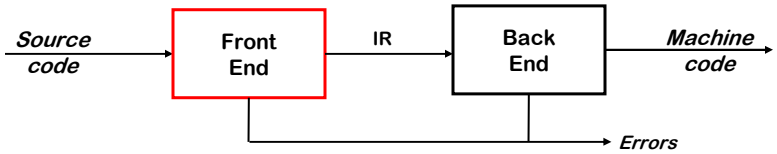
Predictive Parsing

The Front End

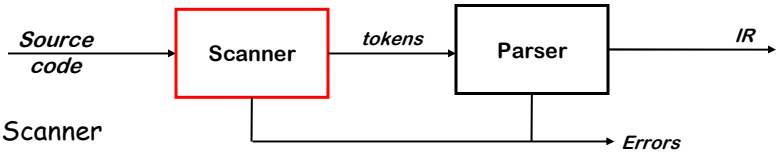


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End - Scanner



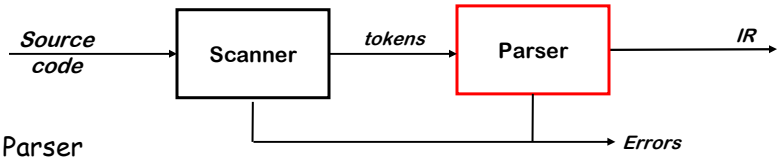
Scanner

- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; *becomes*
    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning

⇒ use a specialized recognizer

The Front End - Parser



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

Roadmap (Where are we?)

In CMSC 330 we studied scanners & parsers

- Specifying tokens
  - → Regular expressions
- Specifying syntax
  - → Context-free grammars

Now we'll look at more advanced parsers

- Predictive top-down parsing
  - → FIRST, FOLLOW, FIRST+
  - → The LL(1) condition
  - → Table-driven LL(1) parsers
- Bottom-up shift-reduce parsers

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 *(predictive parsing)*

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

Parsing Techniques: Top-down parsers

LL(1), recursive descent

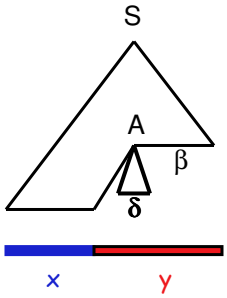
1 input symbol lookahead

construct leftmost deriviation (forwards)

input: read left-to-right

rule $A \rightarrow \delta$

$$S \Rightarrow^*_{lm} x\,A\,\beta \Rightarrow_{lm} x\,\delta\,\beta \Rightarrow^*_{lm} x\,y$$



Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

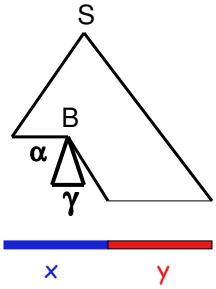
1 input symbol lookahead

construct rightmost deriviation (backwards)

input: read left-to-right

rule $B \rightarrow \gamma$

$$S \Rightarrow^*_{rm} \alpha\,B\,y \Rightarrow_{rm} \alpha\,\gamma\,y \Rightarrow^*_{rm} x\,y$$



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 the fringe of the parse tree matches the input string

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 to the fringe and it doesn't match the fringe, backtrack*
3. *Find the next node to be expanded* *(label ∈ NT)*

- The key is picking the right production in step 1
  → *That choice should be guided by the input string*

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

Predictive Parsing

Basic idea

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

We can try to predict the correct choice by calculating

FIRST(α) sets

The set of tokens that appear as the first symbol in some string that derives from α

That is, $a \in$ FIRST(α) *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

FOLLOW(A) sets

The set of tokens that appear immediately to the right of A in some sentential form

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, *the parser should be able to choose between* α & 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 that derives from α

That is, $a \in$ FIRST(α) *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

The LL(1) Property

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

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

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

This is almost correct
Will also need FOLLOW

The FIRST Set

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

To build FIRST(X) for all grammar symbols X:

1. if X is a terminal (token), FIRST(X) := { X }
2. if $X ::= \varepsilon$, then $\varepsilon \in$ FIRST(X)
3. iterate until no more terminals or ε can be added to any FIRST(X):

   if $X ::= Y_1 Y_2 \ldots Y_k$ then

   $a \in$ FIRST(X) if $a \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \le j < i$

   $\varepsilon \in$ FIRST(X) if $\varepsilon \in$ FIRST(Y_i) for all $1 \le i \le k$

   end iterate

Note: if $\varepsilon \notin$ FIRST(Y_1), then FIRST(Y_i) is irrelevant, for $1 < i$

The FIRST Set

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{a}\,\gamma$, for some γ

To build FIRST(α) for $\alpha = X_1 X_2 \ldots X_n$:

1. $x \in$ FIRST(α) if $x \in$ FIRST(X_i) and $\varepsilon \in$ FIRST(X_j) for all $1 \le j < i$
2. $\varepsilon \in$ FIRST(α) if $\varepsilon \in$ FIRST(X_i) for all $1 \le i \le n$

LL(1) Example – First Sets

Grammar			FIRST Sets: Production	Nonterminals	
Goal	→	Expr	{ num, id }	Goal	{ num, id }
Expr	→	Term Expr'	{ num, id }	Expr	{ num, id }
Expr'	→	+ Expr'	{ + }	Expr'	{ +, -, ε }
	|	- Expr'	{ - }	Term	{ num, id }
	|	ε	{ ε }	Term'	{ *, /, ε }
Term	→	Factor Term'	{ num, id }	Factor	{ num, id }
Term'	→	* Term	{ * }		
	|	/ Term	{ / }		
	|	ε	{ ε }		
Factor	→	num	{ num }		
	|	id	{ id }		

1. if X is a token, FIRST(X) := { X }
2. if X ::= ε, then ε ∈ FIRST(X)
3. if X ::= $Y_1 Y_2 \ldots Y_k$ then
   a ∈ FIRST(X) if a ∈ FIRST(Y_i) …

The FOLLOW Set

For a non-terminal A, define FOLLOW(A) as

FOLLOW(A) := the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set

The FOLLOW Set

To build FOLLOW(X) for all non-terminal X:

1. Place $ in FOLLOW(<*goal*>) // $ = EOF

iterate until no more terminals or ε can be added to any FOLLOW(X):

2. If $A \rightarrow \alpha B \beta$ then
   put {FIRST(β) - ε} in FOLLOW(B)
3. If $A \rightarrow \alpha B$ then
   put FOLLOW(A) in FOLLOW(B)
4. If $A \rightarrow \alpha B \beta$ and $\varepsilon \in$ FIRST(β) then
   put FOLLOW(A) in FOLLOW(B)

LL(1) Example – Follow Sets

Grammar			FIRST Sets	FOLLOW Sets
Goal	→	Expr	Goal { num, id }	{ $ }
Expr	→	Term Expr'	Expr { num, id }	{ $ }
Expr'	→	+ Expr'	Expr' { +, -, ε }	{ $ }
	|	- Expr'	Term { num, id }	{ +, -, $ }
	|	ε	Term' { *, /, ε }	{ +, -, $ }
Term	→	Factor Term'	Factor { num, id }	{ *, /, +, -, $ }
Term'	→	* Term		
	|	/ Term		
	|	**ε**		
Factor	→	num		
	|	id		

1. Place $ in FOLLOW(<*goal*>)
2. If $A \rightarrow \alpha B \beta$ then
   put {FIRST(β) - ε} in FOLLOW(B)
3. If $A \rightarrow \alpha B$ then
   put FOLLOW(A) in FOLLOW(B)
4. If $A \rightarrow \alpha B \beta$ and $\varepsilon \in$ FIRST(β) then
   put FOLLOW(A) in FOLLOW(B)

Predictive Parsing

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), too

Define FIRST$^+$(δ) for rule $A \rightarrow \delta$ as

- FIRST(δ) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(δ)
- FIRST(δ), otherwise

Predictive Parsing

The LL(1) Property

A grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies FIRST$^+$(α) $\cap$ FIRST$^+$(β) = $\varnothing$

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

Question: Can there be two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$ in a *LL(1)* grammar such that $\varepsilon \in$ FIRST(α) and $\varepsilon \in$ FIRST(β)?

Predictive Parsing

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

- Problem: NT A needs to be replaced in next derivation step
- Assume $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with
  $FIRST^+(\beta_1) \cap FIRST^+(\beta_2) \cap FIRST^+(\beta_3) = \varnothing$

```
/* find rule for A */
if (current token ∈ FIRST+ (β1))
   select A → β1
else if (current token ∈ FIRST+(β2))
   select A → β2
else if (current token ∈ FIRST+(β3))
   select A → β3
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. *The other is a table-driven parser table-driven parser.*

LL(1) Parser Example

Is the following grammar LL(1)?

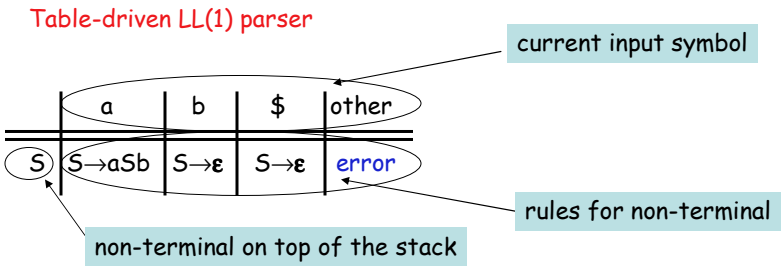
S ::= a S b | **ε**

First(aSb) = { a }
First(**ε**) = { **ε** }

$First^+$(aSb) = { a }
$First^+$(ε) = (First (ε) - { ε }) ∪ Follow (S) = { $, b }

LL(1)? YES, since { a } ∩ { $, b } = ∅

LL(1) Parser Example



Building Table-driven Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- Need an algorithm to build the table

Filling in TABLE[X,y], X ∈ *NT*, y ∈ *T*

- entry is the rule $X ::= \beta$, if y ∈ FIRST+(β)
- entry is error otherwise (can treat empty entry as implicit error)

If any entry is defined multiple times, *G* is not *LL(1)*

This is the *LL(1)* table construction algorithm

LL(1) Skeleton Parser

```
token ← next_token()
push $ onto Stack              // $ used to mark EOF
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = $ and token = $ then
     break & report success (accept)
  else if TOS is a terminal then
     if TOS matches token then
        pop Stack              // recognized TOS
        token ← next_token()
     else report error looking for TOS
  else                         // TOS is a non-terminal
     if TABLE[TOS,token] is A→ B1B2…Bk then
        pop Stack              // get rid of A
        push Bk, Bk-1, …, B1   // in that order
     else report error expanding TOS
  TOS ← top of Stack
```

exit on success

Table-driven *LL(1)* Parser Example

	a	b	$	other
S	S→aSb	S→ε	S→ε	error

How to parse input a a a b b b ?

Describe action as sequence of states

(PDA stack content, remaining input, next action)

PDA stack content: [X, … Z], where Z is the TOS
next actions: rule or next input+pop or error or accept

Table-driven *LL(1)* Parser Example

	a	b	$	other
S	S→aSb	S→ε	S→ε	error

For S→aSb
1. Pop S
2. Push b, push S, push a (i.e., in reverse order)

Stack	Remaining Input	Action
[$, S],	aaabbb$,	S→aSb
[$, b, S, a],	aaabbb$,	next input+pop
[$, b, S],	aabbb$,	S→aSb
[$, b, b, S, a],	aabbb$,	next input+pop
[$, b, b, S],	abbb$,	S→aSB
[$, b, b, b, S, a],	abbb$,	next input+pop
[$, b, b, b, S],	bbb$,	S→ε
[$, b, b, b],	bbb$,	next input+pop
[$, b, b],	bb$,	next input+pop
[$, b],	b$,	next input+pop
[$],	$,	accept

LL(1) Example – LL(1) Table

Grammar		FIRST Sets	FOLLOW Sets	
Goal	→ Expr	{ num, id }	Goal	{ $ }
Expr	→ Term Expr'	{ num, id }	Expr	{ $ }
Expr'	→ + Expr'	{ + }	Expr'	{ $ }
	| - Expr'	{ - }	Term	{ +, -, $ }
	| ε	{ ε }	Term'	{ +, -, $ }
Term	→ Factor Term'	{ num, id }	Factor	{ *, /, +, -, $ }
Term'	→ * Term	{ * }		
	| / Term	{ / }		
	| ε	{ ε }		
Factor	→ num	{ num }		
	| id	{ id }		

For TABLE[X,y], X ∈ NT, y ∈ T
entry is the rule X → β, if y ∈ FIRST+(β)

	num	id	+	-	*	/	$
Goal	Goal → Expr	Goal → Expr					
Expr	Expr → Term Expr'	Expr → Term Expr'					
Expr'			Expr' → + Expr'	Expr' → - Expr'			Expr' → ε
Term	Term → Factor Term'	Term → Factor Term'					
Term'			Term' → ε	Term' → ε	Term' → * Term	Term' → / Term	Term' → ε
Factor	Factor → num	Factor → id					

LL(1) Languages

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Example

$\{a^n 0\ b^n \mid n \geq 1\} \cup \{a^n 1\ b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

Language that Cannot Be LL(1)

Example

$\{a^n 0\ b^n \mid n \geq 1\} \cup \{a^n 1\ b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

$G \rightarrow \underline{a}A\underline{b}$
$\quad \mid \underline{a}B\underline{bb}$
$A \rightarrow \underline{a}A\underline{b}$
$\quad \mid \underline{0}$
$B \rightarrow \underline{a}B\underline{bb}$
$\quad \mid \underline{1}$

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.