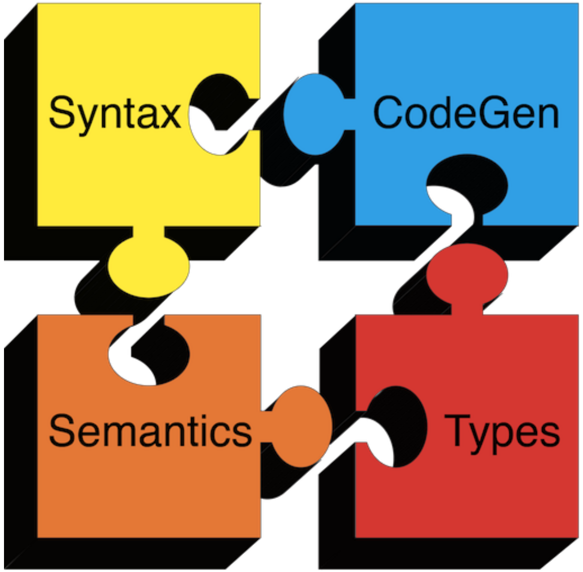


Compilers

Predictive Parsing

Alex Aiken

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking

lookahead
restricted grammars

- Predictive parsers accept LL(k) grammars

left-to-right
left-most derivation
k tokens lookahead
(k=1)

Predictive Parsing

- In recursive descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices

- In LL(1),
  - At each step, only one choice of production

$w\underline{A}\beta$ next input $+$

<u>one</u> $A \rightarrow \alpha$ on input $+$

$w\alpha\beta$

Predictive Parsing

- Recall the grammar

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

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

- We need to left-factor the grammar

Predictive Parsing

- Recall the grammar
  - $E \rightarrow T + E \mid T$
  - $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

$E \rightarrow T X$

$X \rightarrow + E \mid \epsilon$

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

$Y \rightarrow * T \mid \epsilon$

Predictive Parsing

Choose the alternative that correctly left factors “if” statements in the given grammar

EXPR → if BOOL then { EXPR }
  | if BOOL then { EXPR } else { EXPR }
  | …
BOOL → true | false

- EXPR → if true then { EXPR }
  | if false then { EXPR }
  | if true then { EXPR } else { EXPR }
  | if false then { EXPR } else { EXPR }
  | …

- EXPR → EXPR’ | EXPR’ else { EXPR }
  EXPR’ → if BOOL then { EXPR }
  | …
  BOOL → true | false

- EXPR → if BOOL EXPR’
  | …
  EXPR’ → then { EXPR }
  | then { EXPR } else { EXPR }
  BOOL → true | false

- EXPR → if BOOL then { EXPR } EXPR’
  | …
  EXPR’ → else { EXPR } | ε
  BOOL → true | false

Predictive Parsing

- 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

Predictive Parsing

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is int, use production $E \rightarrow T\ X$"

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

- 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

$Y \rightarrow \epsilon$

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

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

Predictive Parsing

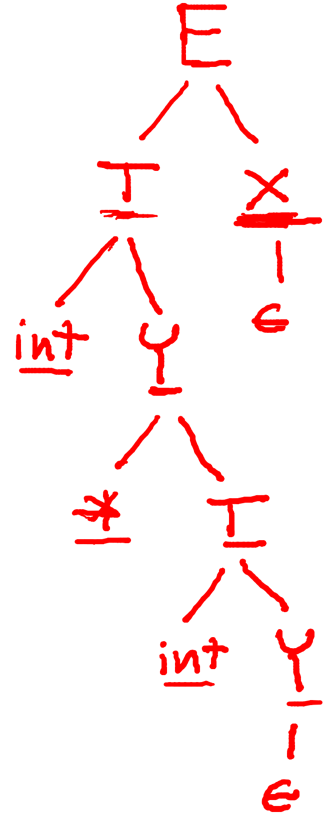
- Method similar to recursive descent, except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]
- A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input & empty stack

Predictive Parsing

```
initialize stack = <S $> and next
                         end of input
repeat
   case stack of
      <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 == < >
```

Predictive Parsing

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



Predictive Parsing

Choose the next parse state given the grammar, parse table, and current state below. The initial string is:

if true then { true } else { if false then { false } } $

	if	then	else	{	}	true	false	$
E	if B then { E } E’				ε	B	B	ε
E’			else { E }		ε			ε
B						true	false	

$E \rightarrow$ if B then { E } E’
| B | ε
$E' \rightarrow$ else { E } | ε
$B \rightarrow$ true | false

	Stack	Input
Current	E’ $	else { if false then { false } } $
○	$	$
○	else { E } $	else { if false then { false } } $
○	E } $	if false then { false } } $
○	else {if B then { E } E’ } $	else { if false then { false } } $