Lecture #8

Syntax Analysis - II

Top Down Parsing

- *A top-down parser starts with the root of the parse tree, labeled 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 labeled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
  2. When a terminal is added to the fringe that doesn´t match the input string, backtrack (Some grammars are backtrack free (*predictive*))
  3. Find the next node to be expanded

- **The key is selecting the right production in step 1**
  - *should be guided by input string*

Recursive Descent Parsing

- Parse tree is constructed
  - From the top level non-terminal
  - Try productions in order from left to right

- Terminals are seen in order of appearance in the token stream.

- When productions fail, backtrack to try other alternatives

- Example:
  - Consider the parse of the string: (int_5)
  - The grammar is : $\mathbf{E \rightarrow T \mid T + E}$

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

Recursive Descent Parsing Algorithm

- TOKEN – type of tokens
  - In our case, let the tokens be: INT, OPEN, CLOSE, PLUS, TIMES
- *next – points to the next input token

- Define boolean functions that check for a match of:
  - A given token terminal

    bool term(TOKEN tok) { return *next++ == tok; }
  - The n^{th} production of a particular non-terminal S:

    bool S_n() { … }
  - Try all productions of S:

    bool S() { … }

Recursive Descent Parsing Algorithm

Functions for non-terminal 'E'
[E → T | T + E]

- For production E → T

```
bool E1() { return T(); }
```

- For production E → T + E

```
bool E2() { return T() && term(PLUS) && E(); }
```

- For all productions of E (with backtracking)

```
bool E() {
    TOKEN *save = next;
    return (next = save, E1())
           || (next = save, E2());
}
```

Recursive Descent Parsing Algorithm

- **Functions for non-terminal T : [T → int | int * T | (E)]**
  - bool T_1() { return term(INT); }
  - bool T_2() { return term(INT) && term(TIMES) && T(); }
  - bool T_3() { return term(OPEN) && E() && term(CLOSE); }
  - bool T() {

```
    TOKEN *save = next;
    return (next = save, T1())
           || (next = save, T2())
           || (next = save, T3());
}
```

Recursive Descent Parsing Algorithm

- To start the parser
  - Initialize *next* to point to first token
  - Invoke *E*()
- Try parsing by hand:
  - **(int$_5$ * int$_7$)**

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {
        TOKEN *save = next;
        return (next = save, E1()) || (next = save, E2());
}
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) &&
                                              T(); }
bool T3() { return term(OPEN) && E() &&
                                    term(CLOSE); }
bool T() {
        TOKEN *save = next;
        return (next = save, T1())
              || (next = save, T2())
              || (next = save, T3());
}
```

Predictive Parsers

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

- Predictive parsers restrict CFGs and accept LL(k) grammars
  - 1^{st} L – The input is scanned from left to right
  - 2^{nd} L – Create left-most derivation
  - K – Number of symbols of look-ahead

- Most parsers work with one symbol of look-ahead [LL(1)]
- Informally, LL(1) has no left-recursion and has been left-factored.

Left Factoring

- Given the grammar:

  $E \rightarrow T + E \mid T$

  $T \rightarrow int \mid int * T \mid (E)$
- Hard to predict which production to use because of common prefixes.
- We need to left-factor the grammar
- The Left-factored grammar:

  $E \rightarrow T\ X$

  $X \rightarrow + E \mid \varepsilon$

  $T \rightarrow (E) \mid int\ Y$

  $Y \rightarrow * T \mid \varepsilon$

LL(1) Parsing Table

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

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

Parsing Table

Next input token

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

Left-most non-terminal

RHS of production to use

Predictive Parsing Algorithm

- For the leftmost non-terminal X
- We look at the next input token a
- Makes use of an explicit parsing table of the form M[X, a]

- An explicit external stack records frontier of the 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 Algorithm

- The parse table entry M[X, a] indicates which production to use if the top of the stack is a non-terminal ‘X’ and the current input token is ‘a’.
- In that case ‘POP X’ from the stack and ‘PUSH’ all the RHS symbols of the production M[X, a] in reverse order.
- Assume that '$' is a special token that is at the bottom of the stack and terminates the input string

```
if X = a = $ then halt
if X = a ≠ $ then pop(x) and ip++
if X is a non terminal
      then if M[X, a] = {X → UVW}
            then begin pop(X); push(W,V,U)
                  end
      else error
```

Predictive Parsing Algorithm

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

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

Another Example

$E \rightarrow T E'$ $\quad$ $E' \rightarrow +T E' \mid \epsilon$

$T \rightarrow F T'$ $\quad$ $T' \rightarrow * F T' \mid \epsilon$

$F \rightarrow (E) \mid id$

	id	+	*	(	)	$
E	**TE'**			**TE'**		
E'		**+TE'**			**Є**	**Є**
T	**FT'**			**FT'**		
T'		**Є**	***FT'**		**Є**	**Є**
F	**id**			**(E)**		

Try this out for the input string:

id + id * id $

Another Example

Stack	Input	Action
E$	id + id * id $	TE’
TE’$	id + id * id $	FT’
FT’E’$	id + id * id $	id
idT’E’$	id + id * id $	terminal
T’E’$	+ id * id $	Є
E’$	+ id * id $	+TE’
+TE’$	+ id * id $	terminal
TE’$	id * id $	FT’
FT’E’$	id * id $	id
idT’E’$	id * id $	terminal
T’E’$	* id $	*FT’
*FT’E’$	* id $	terminal
FT’E’$	id $	id
idT’E’$	id $	terminal
T’E’$	$	Є
E’$	$	Є
$	$	ACCEPT

	id	+	*	(	)	$
E	**TE’**			**TE’**		
E’		**+TE’**			**Є**	**Є**
T	**FT’**			**FT’**		
T’		**Є**	***FT’**		**Є**	**Є**
F	**id**			**(E)**		

Next Lecture

Top-Down Parsing - II