CS536 Lecture 14

Thursday 5 March 2015

Last class:

- Building the selector table

Today:

- SDT for Parsing

Building the Parse Table

FIRST sets: The set of terminals that can begin the strings derivable from a *sequence*. Includes ∈ if that sequence can derive ∈.

FOLLOW sets: The set of terminals that can appear immediately to the right of a nonterminal in a partial derivation.

Algorithm:

```
For each production X → α:
    For each terminal t in FIRST(α):
        put X → α in Table[X][t]

    If ∈ is in FIRST(α):
        For each terminal t in FOLLOW(X):
            Put X → α in Table[X][t]
        If eof is in FOLLOW(X):
            Add X → α to Table[X][eof] as well
```

Note: Blank cells imply a syntax error.

Putting it all together:

- Build FIRST sets for each nonterminal
- Build FIRST sets for each RHS
- Build FOLLOW sets for each nonterminal
- Use FIRST and FOLLOW to fill in parse table

Parse Table Example

Grammar:

$S \rightarrow$ **(** S **)** | **{** S **}** | $\in$

FIRST and FOLLOW:

S :

S:

(S **)**:

{ S **}**:

Parse Table:

	(	**)**	**{**	**}**	**eof**
S					

Parse Table Example 3

Grammar:

$S \rightarrow + S \mid \in$

FIRST and FOLLOW:

S :

S:

$+ S$:

Parse Table:

	+	**eof**
S		

An SDT for Parsing

Do we really need to build the parse tree?

Convert SDT rules to “actions”

“Semantic” stack:

- Pop translations of RHS nonterminals.
- Push computed translation of LHS nonterminal.

Example:

Grammar:	SDT Rules:	SDT Actions:
$E \rightarrow \varepsilon$	E.trans = 0	push 0
| **(** *Expr* **)**	E.trans = E_2.trans+1	E_2.trans = pop; push E.trans+1
| **[** *Expr* **]**	E.trans = E_2.trans	E_2.trans = pop; push E.trans

Action Numbers

Our SDT is bottom-up however …

Example Grammar:

	SDT Rules:	SDT Actions:
E → ε **#1**	E.trans = 0	**#1** push 0
| **(** *Expr* **)** **#2**	E.trans = E_2.trans+1	**#2** E_2.trans=pop;push E.trans+1
| **[** *Expr* **]** **#3**	E.trans = E_2.trans	**#3** E_2.trans=pop;push E.trans

Parse Table:

	(	**)**	**[**	**]**	**eof**
E	(*E*) *#2*	[*E*] *#3*	ε *#1*	ε *#1*	ε *#1*

Input: **(** **[** **]** **)** **eof**

Work Stack Semantic Stack

Placing Action Numbers

Assign numbers to actions:

For nonterminals: For terminals:

Example: Multiply the elements in a list:

```
List  → Val List' #1      #1 LTrans = pop(); vTrans = pop();
                              push(LTrans * vTrans);
List' → Val List' #2      #2 LTrans = pop(); vTrans = pop();
                              push(LTrans * vTrans);
      | ε         #3      #3 push(1)
Val → #4 intlit           #4 push(intlit.value)
```

A Benefit of Action Numbers

Grammar:

```
Expr → Expr - Term   #1
     | Term

Term → #2 INTLIT
```

Transformed grammar:

```
Expr  → Term Expr'
Expr' → - Term #1 Expr'
      | ε

Term → #2 INTLIT
```

Actions:

```
#1: tTrans = pop; eTrans = pop; push(tTrans + eTrans)
#2: push(INTLIT.value)
```

Input: 4 - 2 - 1

Back to ASTs

Actions can construct AST nodes.

Grammar:

```
Expr → Expr - Term    #1
     | Term

Term → #2 INTLIT
```

Transformed grammar:

```
Expr  → Term Expr'
Expr' →  - Term #1 Expr'
       | ε

Term → #2 INTLIT
```

Actions:

```
#1: tTrans = pop; eTrans = pop;
    push(new MinusNode(tTrans, eTrans))
#2: push(new IntLitNode(INTLIT.value))
```

Parse Table:

	INTLIT	**-**	**eof**
Expr	*Term Expr'*		
Expr'		*- Term #1 Expr'*	ε
Term	*#2* **INTLIT**		

Input: `4 - 2 - 1 eof`

Work Stack

Semantic Stack

Take Stock

We covered all procedures up until constructing an AST.

But only for LL(1) grammars.

Turns out LL(1) is fairly restrictive.

Dangling-else grammar:

```
IfStmt → if ( boolExpr ) Stmts
       |  if ( boolExpr ) Stmts else Stmts
```

Big picture

Zoom back out …