CS 536 Announcements for Monday, March 11, 2024

Programming Assignment 3 – due Friday, March 15

Midterm 2 – Thursday, March 21

Last Time
- review grammar transformations
- building a predictive parser
- FIRST and FOLLOW sets

Today
- review parse table construction
- predictive parsing and syntax-directed translation

Next Time
- static semantic analysis

Recap of where we are

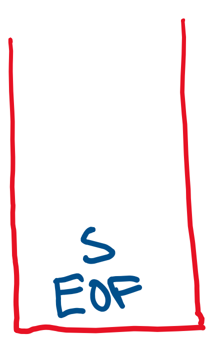
Predictive parser builds the parse tree top-down
- 1 token lookahead
- parse/selector table
- stack tracking current parse tree's frontier

Building the parse table – given production *lhs* → *rhs*, determine what terminals would lead us to choose that production

ie figure out T so that table[lhs][T] = rhs

FIRST(α) = $\{ T \mid (T \in \Sigma \wedge \alpha \Rightarrow^* T\beta) \vee (T = \varepsilon \wedge \alpha \Rightarrow^* \varepsilon) \}$

FOLLOW(a) = $\{ T \mid (T \in \Sigma \wedge s \Rightarrow^* \alpha a T \beta) \vee (T = EOF \wedge s \Rightarrow^* \alpha a) \}$



Current token: D

$S \rightarrow aC$
$\quad \mid ba$

Look at FIRST(aC) & FIRST(ba)

If D is in neither & ba $\Rightarrow^* \varepsilon$, then look at FOLLOW(ba)

FIRST and FOLLOW sets

FIRST(α) for $\alpha = y_1\ y_2 \ldots y_k$

Add FIRST(y_1) – { ε }

If ε is in FIRST($y_{1\ \text{to}\ i-1}$), add FIRST(y_i) – { ε }

If ε is in all RHS symbols, add ε

FOLLOW(a) for x → α a β

If a is the start, add EOF

Add FIRST(β) – { ε }

Add FOLLOW(x) if ε is in FIRST(β) or β is empty

Note that

FIRST sets

- only contain alphabet terminals and ε
- defined for arbitrary RHS and nonterminals
- constructed by started at the beginning of a production

→ at beginning of rhs (for FIRST(lhs))

FOLLOW sets

- only contain alphabet terminals and EOF
- defined for nonterminals only
- constructed by jumping into production

Putting it all together

- Build FIRST sets for each nonterminal
- Build FIRST sets for each production's RHS
- Build FOLLOW sets for each nonterminal
- Use FIRST and FOLLOW sets to fill parse table for each production

Building the parse table

```
for each production x → α {
     for each terminal T in FIRST(α) {
         put α in table[x][T]
     }
     if ε is in FIRST(α) {
         for each terminal T in FOLLOW(x) {
             put α in table[x][T]
         }
     }
}
```

Example

CFG

s → a C | b a
a → A B | C s
b → D | ε

FIRST and FOLLOW sets

	FIRST sets	FOLLOW sets
s	A, C, D	EOF, Follow(a) C
a	A, C	C, EOF
b	D, ε	A, C
s → a C	A, C	
s → b a	D, A, C	
a → A B	A	
a → C s	C	
b → D	D	
b → ε	ε	

Parse table

```
for each production x → α

    for each terminal T in FIRST(α)
        put α in table[x][T]

    if ε is in FIRST(α)
        for each terminal T in FOLLOW(x)
            put α in table[x][T]
```

→ not LL(1)

	A	B	C	D	EOF
s	aC, ba		aC, ba	ba	
a	AB		Cs		
b	ε		ε	D	

Example

CFG

s → (s) | { s } | ε

FIRST and FOLLOW sets

	FIRST sets	FOLLOW sets
s	({ ε	EOF) }
s → (s)	(	
s → { s }	{	
s → ε	ε	

Parse table

```
for each production x → α
    for each terminal T in FIRST(α)
        put α in table[x][T]

    if ε is in FIRST(α)
        for each terminal T in FOLLOW(x)
            put α in table[x][T]
```

	(	)	{	}	EOF
s	(s)	ε	{s}	ε	ε

Parsing and syntax-directed translation

Recall syntax-directed tranlation (SDT)

To translate a sequence of tokens

- build the parse tree
- use translation rules to compute the translation of each non-terminal in the parse tree, bottom up
- the translation of the sequence is the translation of the parse tree's root non-terminal

Goal translation: evaluate expression

CFG:		SDT rules:
expr	→ expr + term	$expr_1$.trans = $expr_2$.trans + term.trans
	| term	expr.trans = term.trans
term	→ term * factor	$term_1$.trans = $term_2$.trans * factor.trans
	| factor	term.trans = factor.trans
factor	→ INTLIT	factor.trans = INTLIT.value
	| (expr)	factor.trans = expr.trans

The LL(1) parser never needed to explicitly build the parse tree
– it was implicitly tracked via the stack.

Instead of building parse tree, give parser a second, ***semantic*** stack

- holds translations of nonterms

SDT ***rules*** are converted to ***actions***

- pop translations of RHS nonterms
- push computed translation of LHS nonterm

translations are popped R-to-L

CFG:		SDT actions:
expr	→ expr + term	tTrans = pop; eTrans = pop; push(eTrans + tTrans)
	| term	✗ tTrans = pop; push(tTrans)
term	→ term * factor	fTrans = pop; tTrans = pop; push(tTrans * fTrans)
	| factor	✗ fTrans = pop; push(fTrans)
factor	→ INTLIT	push(INTLIT.value)
	| (expr)	✗ eTrans = pop; push(eTrans)

useless rules

Parsing and syntax-directed translation (cont.)

Augment the parsing algorithm

- number the actions
- when RHS of production is pushed onto symbol (work) stack, include the actions
- when action is the top of symbol stack, pop & perform the action

CFG:			SDT actions:
expr	→	expr + term #1	#1 tTrans = pop; eTrans = pop; push(eTrans + tTrans)
	|	term	
term	→	term * factor #2	#2 fTrans = pop; tTrans = pop; push(tTrans * fTrans)
	|	factor	
factor	→	#3 INTLIT	#3 push(INTLIT.value)
	|	(expr)	

Placing the action numbers in the productions

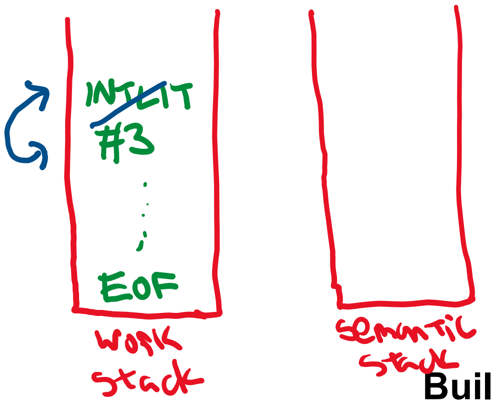
- action numbers go
  - after their corresponding non-terminals
  - before their corresponding terminal

Why? Consider parsing if after

Input: .... INTLIT(17) ...

CFG: factor → INTLIT #3

SDT actions: #3 push(INTLIT.value)



Building the LL(1) parser

1) Define SDT using the original grammar

- write translation rules
- convert translation rules to actions that push/pop using semantic stack
- incorporate action #s into grammar rules

2) Transform grammar to LL(1)

– treating action #s like terminals

3) Compute FIRST and FOLLOW sets

– treating action #s like ε

4) Build the parse table

Example SDT on transformed grammar

Original CFG:

expr → expr + term #1
  | term

term → term * factor #2
  | factor

factor → #3 INTLIT
  | (expr)

Transformed CFG:

expr → term expr'
expr' → + term #1 expr'
  | ε
term → factor term'
term' → * factor #2 term'
  | ε
factor → #3 INTLIT | (expr)

Transformed CFG:

expr → term expr'

expr' → + term #1 expr' | ε

term → factor term'

term' → * factor #2 term' | ε

factor → #3 INTLIT | (expr)

SDT actions:

#1 : tTrans = pop;
eTrans = pop;
push(eTrans + tTrans)

#2 : fTrans = pop;
tTrans = pop;
push(tTrans * fTrans)

#3 : push(INTLIT.val)

Parse table

	+	*	(	)	INTLIT	EOF
expr			term expr'		term expr'	
expr'	+ term #1 expr'			ε		ε
term			factor term'		factor term'	
term'	ε	* factor #2 term'		ε		ε
factor			(expr)		#3 INTLIT	

Input: 5 + 3 * 2 EOF

#2: fTrans = 2
tTrans = 3 → 3*2 = 6

#1: tTrans = 6
eTrans = 5 → 5+6 = 11

What about ASTs?

Push and pop AST nodes on the semantic stack

Keep references to nodes that we pop

Original CFG:

expr → expr + term #1
    | term

term → #2 INTLIT

Transformed CFG:

expr → term expr'
expr' → + term #1 expr'
    | ε
term → #2 INTLIT

SDT actions:

#1 : tTrans = pop;
    eTrans = pop;
    push(new PlusNode(eTrans, tTrans))

#2 : push(new IntLitNode(INTLIT.value))

Parse table:

	INTLIT	+	EOF
expr	term expr'		
expr'		+ term #1 expr'	ε
term	#2 INTLIT