Syntax-Directed Translation & YACC

Dragon: Ch 5. (Just part of it)

Holub: pp 183-192, pp 348-354

Dragon: ch. 4.9

Syntax-Directed Translation

- We cover this topic briefly, mostly for how we use in YACC
- Grammar defines the syntax of a language, but now we want to talk about semantics (meanings)
- Let us first talk about **how to perform semantic evaluation**
  - Approach 1: build the parse tree first and then traverse the tree
  - Approach 2: use parser actions to evaluate & pass semantic values
    - This means that we use the parser as a control structure
    - Advantages of parser doing more than syntax analysis
      - Do not have to construct the parse tree
      - Do not need lots of recursive functions to associate/evaluate values
  - We take the approach 2, which is one kind of syntax-directed translation (SDT)

Semantics

- Semantics in SDT are determined by
  - Semantic values associated with syntactic constructs such as terminals or nonterminals
    - What is semantic value? It depends on syntactic constructs
      - ID: pointer to *struct id*, INT_NUM: integer number, Nonterminals: pointer to *struct decl*, …….
  - Semantic actions taken when reducing a production
    - Evaluate and pass semantic values of reduced RHS
    - Implemented by C code embedded in the RHS of a production, executed at "that" point of parsing
      - Since LR parsing need to process entire RHS before reducing an actual production, we usually embed C code at the end of a RHS
        - It is dangerous to embed action in the middle of a RHS

The Case of YACC

- In YACC
  - Each grammar symbol in a production has a semantic value
    - Expressed using $i notation
  - Action is C code embedded in the RHS of a production
  - Executed at the point it is encountered during parsing
    - Usually at the end of a production

An Example :

```
E :    E '+' E  {$$ = $1 + $3;}
   |   E '*' E  {$$ = $1 * $3;}
   |   num      {$$ = $1;}
;
```

This assumes that the lexer converts and returns the number

Semantic Actions in YACC

- How is semantic evaluation done in YACC?
  - E.g., E : E '+' E {$$ = $1 + $3}
- Run a separate **value stack** in parallel to **state stack**
  - A pseudo variable $i refers to top_of_stack – (|RHS|–i)
  - A pseudo variable $$ refers to the value associated with LHS (nonterminal); it becomes to $1 by default

- $i for i ≤ 0 is also defined.
  - Refer to a stack value BEFORE those that match the current production's RHS
  - Must know the context when applying this action

```
F   :   E   B   '+'   E {$$ = ...}
    ;
B   :   /* empty */ {prev_expr = $0;}
```

$0 refers to value of E in the previous production

  - When is this useful? E.g., declaration: type var

Mid-Rule Actions in YACC

- We sometimes want an action in the middle of RHS

```
A    :    B {$$ = $1 + 1;}
          C {$$ = $2 + $3;}
```

  - Can access component value preceding the action via $i, but cannot refer to forward (i.e., to the right)
  - Action itself counts as a $i thing
  - Can have its own semantic value by assigning to $$
  - Later actions refer to it by $i
  - Cannot assign values to LHS except at the end of the RHS, or by default LHS's value becomes $1

Implementation of Mid-rule Action

- YACC creates a new nonterminal and a production for every mid-rule action.

```
A   :   B {$$ = $1 + 1;}
        C {$$ = $2 + $3;}
```

- Might transform the above as follows:

```
A   :   B M1 C M2{$$ = $4;}
M1  :   /* empty */ {$$ = $0 + 1;}
M2  :   /* empty */ {$$ = $-1 + $0;}
```

- Real YACC does the following

```
A   :   B M1 C {$$ = $2 + $3;}
M1  :   /* empty */ {$$ = $0 + 1;}
```

Conflicts due to Mid-Rule Actions

- Mid-rule actions might lead to conflicts that were not present in the original grammar

```
Example:
    blk   :   BEGIN decls stmts END
          |   BEGIN stmts END
          ;
```

- If we want to do some work to prepare for local variable spaces in the first production

Shift/Reduce Conflicts

- We might want to add mid-rule action as follows:

```
blk   :   {prepare_local_vars();} BEGIN decls stmts END
      |   BEGIN stmts END
      ;
```

- Then, YACC converts the grammar as follows:

```
blk   :   M BEGIN decls stmts END
      |   BEGIN stmts END
      ;
M     :   /* empty */ {prepare_local_vars();}
```

- However, this leads to a shift/reduce conflict

Reduce/Reduce Conflicts

- If we add mid-rule actions differently as follows:

```
blk    :    {prepare_local_vars();} BEGIN decls stmts END
       |    {prepare_local_vars();} BEGIN stmts END
       ;
```

- Then, YACC converts the grammar as follows:

```
blk    :    M1 BEGIN decls stmts END
       |    M2 BEGIN stmts END
       ;
M1     :    /* empty */ {prepare_local_vars();}
M2     :    /* empty */ {prepare_local_vars();}
```

- However, this leads to reduce/reduce conflict

Possible Solutions

- Add the mid-rule action after BEGIN

```
blk   :   BEGIN {prepare_local_vars();} decls stmts END
      |   BEGIN stmts END
      ;
```

- Another solution

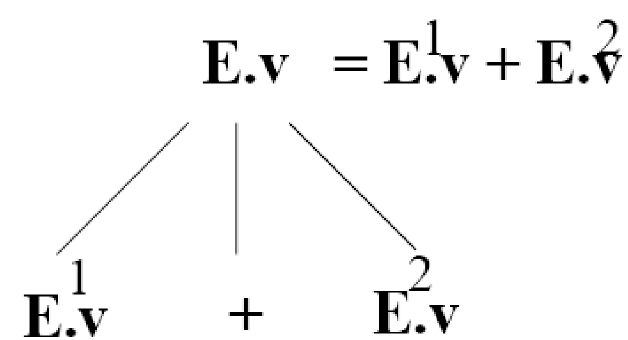
```
blk   :   M BEGIN decls stmts END
      |   M BEGIN stmts END
      ;
M     :   /* empty */ {prepare_local_vars();}
```

Attributes

- **Attributes**: semantic value associated with a node in a parse tree (e.g., $i in YACC)
  - Type, numeric value, string, pointer to C structure, etc.
  - Two types: synthesized and inherited
- Synthesized attribute: value is determined by the children of a node
  - straightforward for bottom-up parsing

E : E '+' E {$$1 = $1 + $3}

Synthesized Attribute

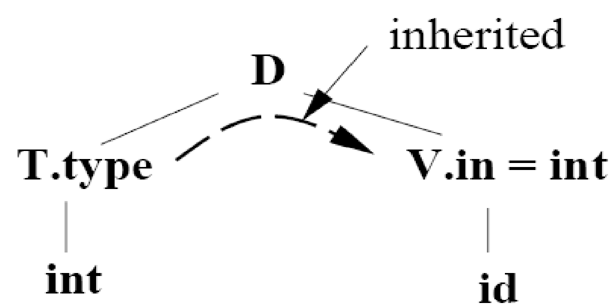


- Inherited attributes: value is passed down from parent or a sibling of a node
  - Example: simple variable declaration rule

```
D  :  T  V ;
T  :  int | real;
V  :  id { add_type( $0, $1) }
   :
```

type attribute

id attribute



- Implementation of inherited attributes
  - Save in a global variable (problem: nested calls)
  - Use value stack and negative attributes (in YACC)

Augmented and Attributed Grammar

- Augmented grammar:
  - Semantic actions are placed in the grammar itself
  - The position of an action determines when it is executed
- Attribute grammar
  - A grammar to which attributes are attached
  - Attributes help to specify code generation actions in greater detail than with an augmented grammar alone
- Augmented, attributed grammar
  - An attributed grammar augmented with actions

An Example Grammar

Attributed, Augmented Grammar

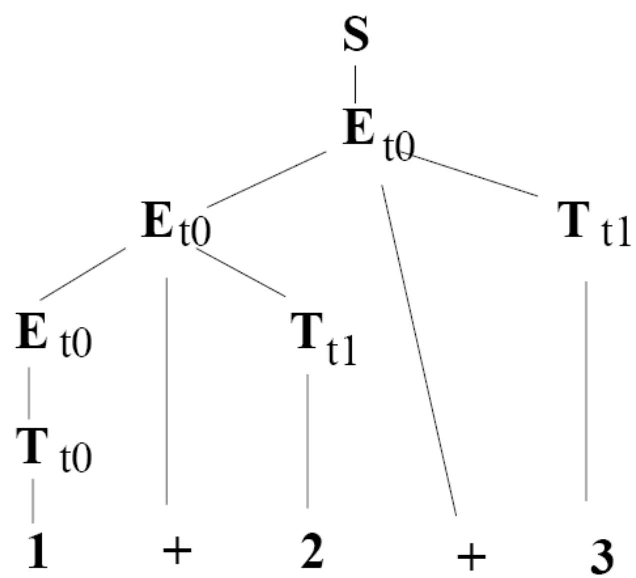
$S_t \to E_t$ { gen(" Answer = %s", E.t); }

$E_t \to E_t + T_t$ { gen(" %s += %s", E.t, T.t); free(T.t); E.t = E.t; }

$E_t \to T_t$ { E.t = T.t}

$T_t \to NUM$ { T.t = new_name(); gen("%s = %s", T.t, yytext); }

Attributed Parse Tree



Code Generation

t0 = 1

t1 = 2

t0 += t1

t1 = 3

t0 += t1

Answer = t0

YACC Specification and Value Stack

S -> E	{gen("Answer = %s", $1)}
E -> E + T	{gen("%s += %s", $1, $3); $$ = $1;}
E -> T	{$$ = $1;}
T -> NUM	{name = new_name(); gen("%s = $s", $$=name, yytext);}

Value Stack

1

$	

2

$	NUM

3

$	T
	t0

t0 = 1

4

$	E
	t0

5

$	E	+
	t0	

6

$	E	+	NUM
	t0		

7

$	E	+	T
	t0		t1

t1 = 2

8

$	E
	t0

t0 += t1

9

$	E	+	NUM
	t0		

10

$	E	+	T
	t0		t1

t1 = 3

11

$	E
	t0

t0 += t1

12

$	S
	t0

Answer = t0