TDDB29 Compilers and Interpreters

TDDB44 Compiler Construction

Attribute Grammars

Christoph Kessler, IDA,
Linköpings universitet, 2007.

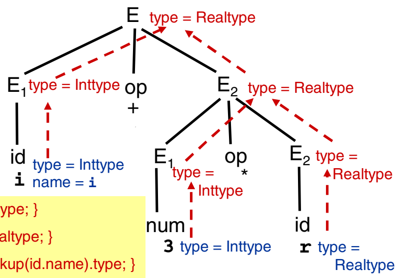
Attribute Grammar

Extended context-free grammar (CFG):

- **Attribute(s)** (value fields) for each nonterminal
- **Semantic rule(s)** for each production
  - imperative or equational computation on attributes
  - executed at reduce (LR parsing) or expand (LL parsing)
- **Inherited Attributes**
  - Information propagated from left to right in a production and downwards in a parse tree
  - E.g., type in declarations, addresses of variables
- **Synthesised Attributes**
  - Information propagated from right to left in a production and upwards in a parse tree
  - E.g., value of expressions, type of expressions, transl. to internal form

Attribute Grammar Example 1
Semantic Analysis – Type Inference

- Given: Attribute Grammar, Parse tree for string `i+3*r`
- Compute: Type for each subexpression (nonterminal)



E → num	{ E.type = Inttype; }
E → num . num	{ E.type = Realtype; }
E → id	{ E.type = lookup(id.name).type; }
E → E_1 op E_2	{ E.type = ... *(see next page)* }

(cont.)

- Attribute grammar for syntax-directed type checking

E → num	{ E.type = Inttype; }
E → num . num	{ E.type = Realtype; }
E → id	{ E.type = lookup(id.name).type; }
E → E_1 op E_2	{ E.type = (E_1.type == Inttype && E_2.type == Inttype)? Inttype : (E_1.type == Inttype && E_2.type == Realtype || E_1.type == Realtype && E_2.type == Inttype || E_1.type == Realtype && E_2.type == Realtype) ? Realtype : error("Type error"), Notype; }

type is a synthesised attribute:
information flows right-to-left, bottom-up

(cont.)

- Attribute grammar extended for assignment statement with implicit type conversion

```
...             ...
E → E1 op E2    { E.type = ... }
...
S → V := E      { if (V.type == E.type)
                     ... // generate code directly according to type
                  else
                     if (V.type == Inttype && E.type == Realtype)
                        error("Type error");
                     else
                        if (V.type == Realtype && E.type == Inttype)
                           // Code generation / evaluation with type conversion:
                           E.value = ... ;
                           V.value = ConvertIntToReal( E.value );
                        }
                }
```


Attribute Grammar Example 2:
Intermediate Code Generation

- Given: Attribute grammar G
- Translate expressions in the language over G(E) to intermediate code in postfix notation

E → E_1 + E_2	{ E.code = concat(E_1.code, E_2.code, "+"); }
| E_1 – T	{ E.code = concat(E_1.code, T.code, "-"); }
| T	{ E.code = T.code; }
T → '0'	{ T.code = "0"; }
| '1'	{ T.code = "1"; }
...	...
| '9'	{ T.code = "9"; }

Attribute grammar example 3: Calculator (an interpreter of expressions)

- Semantic rules calculate the value of an arithmetic expression without generating any intermediate code
- Synthesised attribute N.val for each nonterminal N

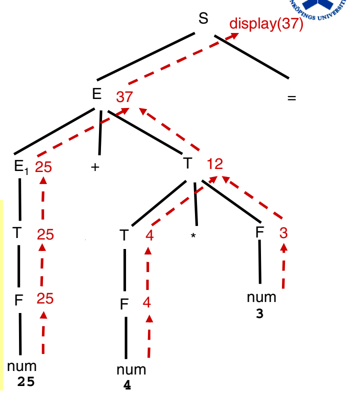
$S \rightarrow E =$	{ display(E.val); }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val); }
$\mid T$	{ E.val = T.val; }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val); }
$\mid F$	{ T.val = F.val; }
$F \rightarrow (E)$	{ F.val = E.val; }
$\mid$ num	{ F.val = num.val; }

value of integer-constant token num as computed by the scanner

(cont.)

- Calculator input:
  `25 + 4 * 3 =`

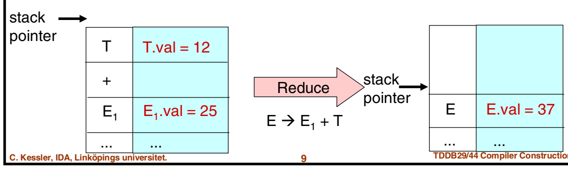
$S \rightarrow E =$	{ display(E.val); }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val); }
$\mid T$	{ E.val = T.val; }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val); }
$\mid F$	{ T.val = F.val; }
$F \rightarrow (E)$	{ F.val = E.val; }
$\mid$ num	{ F.val = num.val; }



Implementation of Attribute Grammars

- **In a LR parser:**
  - Semantic stack in parallel with the parse stack (common stack pointer)
    - Each entry can store all attributes of a nonterminal
  - When performing a reduction $[A \rightarrow \beta_1 \beta_2 \ldots \beta_k .]$
    - calculate all attributes attr by

      A.attr = f (β_1.attr, ..., β_k.attr)



Implementation of Attribute Grammars

- **In a Recursive Descent Parser:**
  - Recall: One procedure for each nonterminal
  - **Interpretation**:
    - Add a parameter for each attribute
      - implicit semantic stack
      - parameters for synthesised attributes to be passed by reference
  - **Code generation:**
    - Write the translated code to a memory buffer or file or return a pointer to generated code block to caller

Example: Calculator for Recursive Descent

LL(1) grammar for calculator:

$S \rightarrow E =$	{ display(E.val); }
$E \rightarrow T_1$	{ E.val = T_1.val; }
$[+ T_2]$	{ E.val = T_1.val + T_2.val; }
$T \rightarrow F_1$	{ T.val = F_1.val; }
$[* F_2]$	{ T.val = F_1.val + F_2.val; }
$F \rightarrow (E)$	{ F.val = E.val; }
$\mid$ num	{ F.val = num.val; }

```
void E ( int *E_val )
{
   int T1_val, T2_val;
   T ( & T1_val );
   *E_val = T1_val;
   while (token == '+') {
       scan();
       T ( & T2_val );
       *E_val += T2_val );
   }
}
```


TDDB29 Compilers and Interpreters

TDDB44 Compiler Construction

Syntax-directed translation of assignment statements and arithmetic expressions into quadruples

using a bottom-up approach

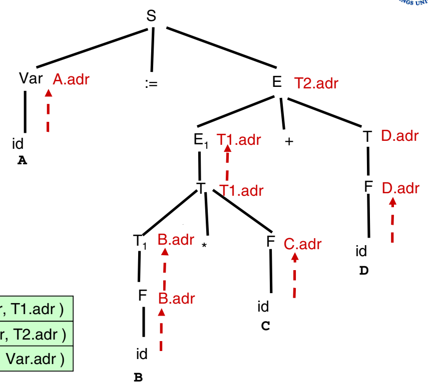
Christoph Kessler, IDA, Linköpings universitet, 2007.

Generating quadruples

op　opnd1　res

1. S → Var := E	{ GEN(ASGN, E.adr, 0, Var.adr); }
2. E → E_1 + T	{ temp = gen_tempvar(); GEN(ADD, E1.adr, T.adr, temp); E.adr = temp; }
3. | T	{ E.adr = T.adr; }
4. T → T_1 * F	{ temp = gen_tempvar(); GEN(MUL, T_1.adr, F.adr, temp); T.adr = temp; }
5. | F	{ T.adr = F.adr; }
6. F → (E)	{ F.adr = E.adr; }
7. | id	{ F.adr = lookup(id.name); }
8. Var → id	{ Var.adr = lookup(id.name); }

Generating quadruples for A := B * C + D



(MUL, B.adr, C.adr, T1.adr)
(ADD, E1.adr, T.adr, T2.adr)
(ASGN, E1.adr, 0, Var.adr)

Generating quadruples for control structures Example: IF-THEN-ELSE

- S → **if** E **then** S_1 **else** S_2

in: <Quadruples for temp := E>
p: (JEQZ, temp, q+1, 0) // jump over S_1 if false
<Quadruples for S_1>
q: (JUMP, r, 0, 0)
q+1: <Quadruples for S_2>
r: …

- Problem: jump target quadruple indices q+1, r are unknown when the jumps are generated
- Solution: factorise the grammar, store jump index in attribute

Generate quadruples for if-then-else (2)

- Factorised grammar:
  1. <ifstmt> ::= <truepart> S_2
  2. <truepart> ::= <ifclause> S_1 **else**
  3. <ifclause> ::= **if** E **then**

Attributes:

addr = address to the symbol table entry for result of E

quad = quadruple number

Generate quadruples for if-then-else (3)

3. <ifclause> ::= **if** E **then**

```
{ <ifclause>.quad = currentquad + 1;
  // save address p of jump over S1 for later in <ifclause>.quad
  GEN ( JEQZ, E.addr, 0, 0 );
  // jump to S2. Target q+1 not known yet.
}
```

2. <truepart> ::= <ifclause> S_1 **else**

```
{ <truepart>.quad = currentquad + 1;
  // save address q of jump over S2 for later
  GEN ( JUMP, 0, 0, 0 );
  // jump over S2. Target r not known yet.
  QUADRUPLE[ <ifclause>.quad ][ 2 ] = currentquad + 1;
  // backpatch JEQZ target to q+1
}
```

3. <ifstmt> ::= <truepart> S_2

Generate quadruples for if-then-else (4)

3. <ifclause> ::= **if** E **then**
   …
2. <truepart> ::= <ifclause> S_1 **else**

```
{ <truepart>.quad = currentquad + 1;
  // save address q of jump over S2 for later
  GEN ( JUMP, 0, 0, 0 );
  // jump over S2. Target r not known yet.
  QUADRUPLE[ <ifclause>.quad ][ 2 ] = currentquad + 1;
  // backpatch JEQZ target to q+1
}
```

1. <ifstmt> ::= <truepart> S_2

```
{ QUADRUPLE[ <truepart>.quad ][ 1 ] = currentquad + 1;
  // backpatch JUMP target to (r-1)+1
}
```

Similarly: while statement, repeat statement …