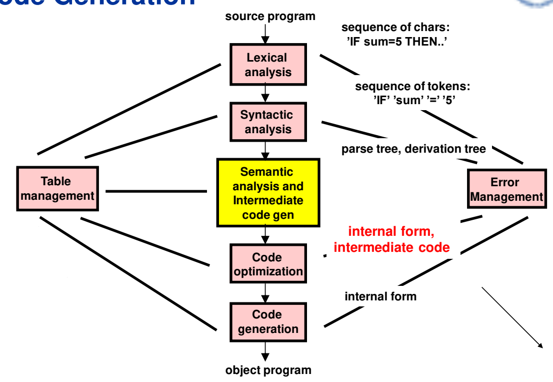
TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

Semantic Analysis and Intermediate Code Generation

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2014

Semantic Analysis and Intermediate Code Generation



Semantic Analysis and Intermediate Representations

- The task of this phase is to check the "static semantics“ and generate the internal form of the program.
- **Static semantics**
  - Check that variables are defined, operands of a given operator are compatible, the number of parameters matches the declaration etc.
  - Formalism for static semantics?
- **Internal form**
  - Generation of good code cannot be achieved in a single pass – therefore the source code is first translated to an internal form.

Methods/Formalisms in Compiler Phases?

- **Which methods / formalisms are used in the various phases during the analysis?**
  1. Lexical analysis: *RE (regular expressions)*
  2. Syntax analysis: *CFG (context-free grammar)*
  3. Semantic analysis and intermediate code generation: *(syntax-directed translation)*

Why not the Same Formalism Everywhere?

Why not use the same formalism (formal notation) during the whole analysis?

- REs are too weak for describing the *language's syntax and semantics.*
- Both *lexical features and syntax of a language can* be described using a CFG. Everything that can be described using REs can also be described using a CFG.
- A CFG can not describe *context-dependent (static semantics) features of a language. Thus there is a* need for a stronger method of ***semantic analysis*** and the *intermediate code generation phase.*

  Syntax-directed translation is commonly used in this phase.

Use of Context Free Grammars vs Regular Expressions?

- **Follow-up questions:**
  - Why are lexical and syntax analysis divided into two different phases?
  - Why not use a CFG instead of REs in lexical descriptions of a language?
- **Answers:**
  - Simple design is important in compilers. Separating lexical and syntax analysis simplifies the work and keeps the phases simple.
  - You build a simple machine using REs (i.e. a scanner), which would otherwise be much more complicated if built using a CFG.

Syntax-Directed Translation in Semantics Phase

The first method we present for the semantics phase is **syntax-directed translation.**
Goal 1: **Semantic analysis:**

- a) Check the program to find semantic errors, e.g. type errors, undefined variables, different number of actual and formal parameters in a procedure, ...
- b) Gather information for the code generation phase, e.g.

```
var a: real;
b: integer
begin
a:= b;
...
```

generates code for the transformation:
`a := IntToReal(b);` // Note: IntToReal is a function for changing integers to a floating-point value.

Goal: Intermediate Code Generation

- Another representation of the source code is generated, a so-called intermediate code representation
- Generation of intermediate code has, among others, the following advantages:

  The internal form is:

  + machine-independent

  + not profiled for a certain language

  + suitable for optimization

  + can be used for interpreting

Examples of Internal/Intermediate forms

- **Internal forms**
  - Infix notation
  - Postfix notation (reverse Polish notation, RPN)
  - Abstract syntax trees, AST
  - Three-address code
  - Quadruples
  - Triples
- **Infix notation**
  - Example:

    a := b + c * (d + e)
  - Operands are between the operators (binary operators). Suitable notation for humans but not for machines because of priorities, associativities, parentheses.

Postfix Notation

Postfix notation
(Also called reverse Polish notation)

- Operators come after the operands.
- No parentheses or priority ordering required.
- Stack machine, compare with an HP calculator.
- Operands have the same ordering as in infix notation.
- Operators come in evaluation order.
- Suitable for expressions without conditions (e.g. if ....)

Examples and comparison:

Infix	Postfix
a + b	**a b +**
a + b * c	**a b c * +**
(a + b) * c	**a b + c ***
a + (-b - 3 * c)	**a b @ 3 c * - +**

Here @ denotes unary minus

Evaluation of Postfix Notation

- Given **an arithmetic expression in reverse Polish (Postfix)** notation it is easy to evaluate directly from left to right.
  - Often used in interpreters.
  - We need a **stack for storing intermediate results.**
- If numeric value:
  - Push the value onto the stack.
- If identifier:
  - Push the value of the identifier (r-value) onto the stack.
- If binary operator:
  - Pop the two uppermost elements , apply the operator to them and push the result.
- If unary operator:
  - Apply the operator directly to the top of the stack.
  - When the expression is completed, the result is on the top of the stack.

Example Evaluation of Postfix Notation

- Example: evaluate the postfix expression below.

a b @ 3 c * - +

Given that a = 34, b = 4, c = 5
corresponding infix notation: a + (-b - 3 * c)

Step	Stack	Input
1	`-|`	`ab@3c*-+ |-`
2	`-|34`	`b@3c*-+ |-`
3	`-|34 4`	`@3c*-+ |-`
4	`-|34 -4`	`3c*-+ |-`
5	`-|34 -4 3`	`c*-+ |-`
6	`-|34 -4 3 5`	`*-+ |-`
7	`-|34 -4 15`	`-+ |-`
8	`-|34 -19`	`+ |-`
9	`-|15`	`|-`

Extending Polish/Postfix Notation Assignment Statement

- **Assignment**
  - := binary operator,
  - lowest priority for infix form,
  - uses the l-value for its first operand
- Example:

  x := 10 + k * 30

  ⇓

  x 10 k 30 * + :=

Extending Polish/Postfix Notation Conditional Statement

- We need to introduce the unconditional jump, JUMP, and the conditional jump, JEQZ, Jump if EQual to Zero, and also we need to specify the jump location, LABEL.

```
L1 LABEL (or L1: )
<label> JUMP
<value> <label> JEQZ
(value = 0 ⇒ false, otherwise ⇒ true)
```

Example 1:

```
IF <expr> THEN <statement1> ELSE <statement2>
```

gives us

```
<expr> L1 JEQZ <statement1> L2 JUMP L1: <statement2> L2:
```

where L1: stands for L1 LABEL

Example 2, Postfix Notation for If-then-Else Statements

```
if a+b then
  if c-d then
    x := 10
  else y := 20
else z := 30;
```

gives us

```
a b + L1 JEQZ
c d - L2 JEQZ
x 10 := L3 JUMP
L2: y 20 := L4 JUMP
L1: z 30 := L3: L4:
```


Small Postfix Notation Exercise

if a+b then x :=4 else x:=33	a b + L1 JEQZ x 4 := L2 JUMP L1: x 33 := L2: ...

Representing While Suitable Data Structure for Postfix Code

while *<expr>* ***do*** *<stat>*

gives us

L2: *<expr> L1 JEQZ <stat> L2 JUMP L1:*

Exercise

Translate the **repeat** and **for** statements to postfix notation.

Suitable data structure for postfix code

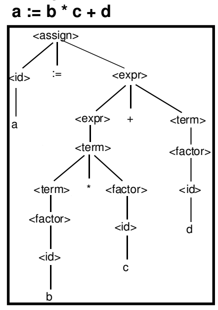
An *array where label corresponds to index.*

Array Elements:

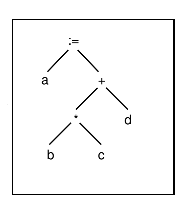
- Operand – pointer to the symbol table.
- Operator – a numeric code, for example, which does not collide with the symbol table index.

Abstract Syntax Trees (AST)

- ASTs are a reduced variant of parse trees. A parse tree contains redundant information, see the figure below.
- **Example: Parse tree for a := b * c + d**



Abstract syntax tree for a := b * c + d :



Properties of Abstract Syntax Trees

- Advantages and disadvantages of abstract syntax trees
  - \+ Good to perform optimization on
  - \+ Easy to traverse
  - \+ Easy to evaluate, i.e. suitable for interpreting
  - \+ *unparsing (prettyprinting) possible via inorder traversal*
  - \+ *postorder traversing gives us postfix notation!*
  - − Far from machine code

Stay with as far as possible

Three-address Code and Quadruples

Three-address code

- op: = +, -, *, /, :=, JEQZ, JUMP, [ ]=, =[ ]

```
z  := x  op  y
↑       ↑      ↑
addr1 addr2  addr3
```

Quadruples

- Form:

  Example: Assignment statement
  A := B * C + D

- gives us the quadruples

  T1 := B * C

  T2 := T1 + D

  A := T2

- T1, T2 are temporary variables.
- The contents of the table are references to the symbol table.

Quadruples:

```
op arg1 arg2 res
```

op	arg1	arg2	res
*	B	C	T1
+	T1	D	T2
:=	T2		A

Control Structures Using Quadruples

- Example:
  if a = b
  then x := x + 1
  else y := 20;

Quad-no	op	arg1	arg2	res
1	=	a	b	T1
2	JEQZ	T1		(6) †
3	+	x	1	T2
4	:=	T2		x
5	JUMP			(7) †
6	:=	20		y
7				

† The jump address was filled in later as we can not know in advance the jump address during generation of the quadruple in a phase. We reach the addresses either during a later pass or by using syntax-directed translation and filling in when these are known. This is called **backpatching.**

Procedure call

- Example:f(a1, a2, ..., an)

Quad-no	op	arg1	arg2	res
1	param	a1		
2	param	a2		
...	...	...		
n	:=	an		
n+1	call	f	n	

- Example: READ(X)

Quad-no	op	arg1	arg2	res
1	param	X		
2	call	READ	1	

- Example: WRITE(A*B, X+5)

Quad-no	op	arg1	arg2	res
1	*	A	B	T1
2	+	X	5	T2
3	param	T1		
4	param	T2		
5	call	WRITE	2	

Array-reference

A[I] := B

[ ]= is called l-value, specifies the address to an element. In l-value context we obtain storage adress from the value of T1.

Quad-no	op	arg1	arg2	res
1	[]=	A	I	T1
2	:=	B		T1

B := A[I]

=[ ] is called r-value, specifies the value of an element

Quad-no	op	arg1	arg2	res
1	=[]	A	I	T2
2	:=	T2		B

Quadruples vs triples
Triples (also called two-address code)

Triples Form:

- *Example: A * B + C*
- No temporary name!

Triple-no	op	arg1	arg2
1	*	A	B
2	+	(1)	C

Quadruples:

- − Temporary variables take up space in the symbol table.
- \+ Good control over temporary variables.
- \+ Easier to optimize and move code around.

Triples:

- − Know nothing about temporary variables.
- \+ Take up less space.
- − optimization by moving code around is difficult; in this case indirect triples are used.

Methods for Syntax-Directed Translation 1. Attribute Grammars

There are two main methods:

1. Attribute grammars, *'attributed translation grammars'*

- Describe the translation process using
  - a) CFG
  - b) a number of attributes that are attached to terminal and nonterminal symbols, and
  - c) a number of semantic rules that are attached to the rules in the grammar which calculate the value of the attribute.

2. Syntax Directed Translation Scheme

Describe the translation process using:

a) a CFG

b) a number of semantic operations

e.g. a rule: **`A → XYZ {semantic operation}`**

- Semantic operations are performed:
  - when **reduction** occurs (bottom-up), or
  - during **expansion** (top-down).
- This method is a more procedural form of the previous one (contains implementation details), which explicitly show the evaluation order of semantic rules.

Example 1: Translation Schema for *Semantic Analysis*

- **Intuition**: Attach **semantic actions** to syntactic rules to perform *semantic analysis and intermediate code generation.*
- Part of CFG, variable declarations of a language with non-nested blocks.
- The text in *{} stands for a* ***description of the semantic analysis*** for book-keeping of information on symbols in the symbol table.

<decls> → ...
<decl> → var <name-list> : <type-id>
    {Attach the type of <type-id> to all id in <name-list>}
<name-list> → <name-list> , <name>
    {Check that name in <name-list> is not duplicated, and check that name has not been declared previously}
<name-list> → <name>
    {Check that name has not been declared previously}
<type-id>→ "ident"
    {Check in the symbol table for "ident", return its index if it is already there, otherwise error: unknown type.}
<name>→ "ident"
    {Update the symbol table to contain an entry for this "ident"}

Example 2: Translation Schema *Intermediate Code Generation*

Translation of **infix** notation **to postfix** notation in a bottom-up environment.

Translation of the input string:

a + b * d

becomes in postfix:

a b d * +

See the parse tree on the coming page:

Productions	**Semantic operations**
1 E → E1 + T	{print('+')}
2	T	. . .
3 T → T1 * F	{print('*')}
4	F	. . .
5 F → (E)	. . .
6	id	{print(id)}

Translation Schema *Intermediate Code Generation*, Implementation in LR Case

The parser routine:

```
void parser();
{
  while not done {
   switch action {
    case shift:
    ...
    case reduce:
     semantic(ruleNo);
    ...
   }/* switch */;
  }/* while */;
}/* parser */;
```

The semantic routine:

```
void semantic(int ruleNo);
{
  switch ruleNo {
   case 1: print('+');
   case 3: print('*');
   case 6: print(id);
  };
};
```

Productions	**Semantic operations**
1 E → E1 + T	{print('+')}
2	T	. . .
3 T → T1 * F	{print('*')}
4	F	. . .
5 F → (E)	. . .
6	id	{print(id)}

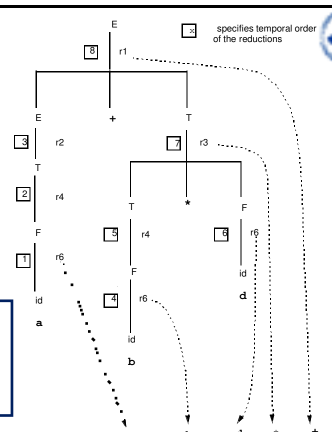
Parse Tree of Translation to Postfix Code

Translation of the input string:

a + b * d

to postfix:

a b d * +



Productions	**Semantic operations**
1 E → E1 + T	{print('+')}
2	T	. . .
3 T → T1 * F	{print('*')}
4	F	. . .
5 F → (E)	. . .
6	id	{print(id)}

TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

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

using a bottom-up approach

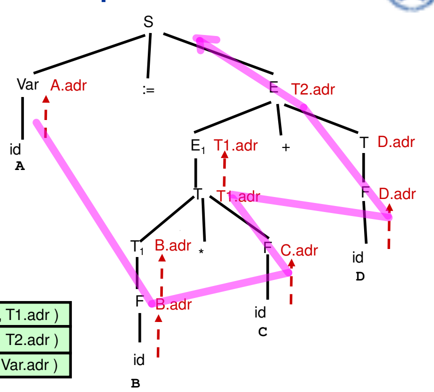
Peter Fritzson, Christoph Kessler, IDA, Linköpings universitet, 2014

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
  - Jump to S_2 if E is false/zero
  - After S_1 jump to after S_2
- Problem: jump target quadruple indices q+1, r are unknown when the jumps are generated
- Solution: factorise the grammar, store jump index in attribute `quad`

Index	Quadruple Table
in:	Quadruples for ....
	.... temp := E
p:	<JEQZ, temp, q+1, 0>
	Quadruples for ...
	... statement S1
q:	<JUMP, r, 0, 0>
q+1: (L1:)	Quadruples for ...
	... statement S2
r: (L2:)	...

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 S_1 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 S_2 for later
GEN (JUMP, 0, 0, 0);
// jump over S_2. 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 S_2 for later
   GEN (JUMP, 0, 0, 0);
   // jump over S_2. 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 …

Generate Quadruples for a while statement

```
WHILE <E> DO <S>
in: quadruples for Temp := <E>
p:  JEQZ Temp q+1 Jump over <S> if <E> false
    quadruples for <S>
q:  JUMP in Jump to the loop-predicate
q+1: ...
```

The grammar factorises on:

```
1. <while-stat> ::= <while-clause> <S>
2. <while-clause>::= <while> <E> DO
3. <while> ::= WHILE
```

An extra attribute, NXTQ, must be introduced here. It has the same meaning as QUAD in the previous example.

`3. {<while>.QUAD ::= NEXTQUAD}`

Rule to find start of <E>

`2. {<while-clause>.QUAD := <while>.QUAD;`

Move along start of <E>

`<while-clause>.NXTQ := NEXTQUAD;`

Save the address to the next quadruple.

`GEN(JEQF, <E>.ADDR, 0, 0)`

Jump position not yet known! }

`1. {GEN(JUMP, <while-clause>.QUAD,0,0);`

Loop, i.e. jump to beginning <E>

`QUADR[<while-clause>.NXTQ,3]:=NEXTQUAD`

(**backpatch**) Position at the end of <S> }

Generate quadruples for:

```
if 1+3
  then 4*5+6
else 2
```

Small Quadruple Generation Exercise

	OP	A1	A2	RES
1	+	1	3	T1
2	JEQZ2	T1		FOO
3	x	4	5	T2
4	+	T2	6	X
5	JMP			FEE
FOO	:=	2		Y
FEE				

(Dbl check)

TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

Attribute Grammars

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2014

Attribute Grammar

Extended context-free grammar (CFG):

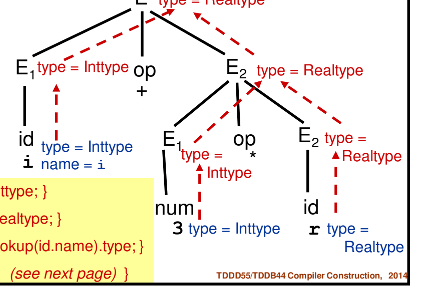
- **Attribute(s)** (value fields) for each nonterminal
- **Semantic rule(s)** for each production
  - equational computation on attributes
  - executed at ***reduce*** (LR parsing) or ***expand*** (LL parsing)

- **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
- **Synthesized 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)

Synthesized attribute: *type*



Grammar rules:	*Semantic rules:*
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

$E \rightarrow$ num	{ E.type = Inttype; }
$E \rightarrow$ num . num	{ E.type = Realtype; }
$E \rightarrow$ id	{ E.type = lookup(id.name).type; }
$E \rightarrow E_1$ 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 from integer to Real

```
...                 ...
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
- For example: 2+3-5 is translated to: 23+5- or 235-+ depending on parse tree
- The attribute *code* is attached to all nonterminals in the grammar
- A semantic rule attached to each grammar rule

$E \rightarrow E_1 + E_2$	{ 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; }
$T \rightarrow$ '0'	{ T.code = "0"; }
| '1'	{ T.code = "1"; }
...	...
| '9'	{ T.code = "9"; }

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

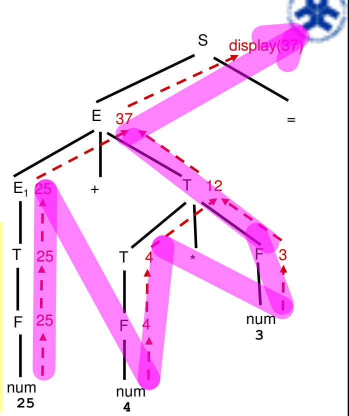
- Semantic rules calculate the value of an arithmetic expression without generating any intermediate code
- Semantic rules execute at grammar rule reductions (LR)
- Synthesised attribute N.val for each nonterminal N

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

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

(cont.)

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

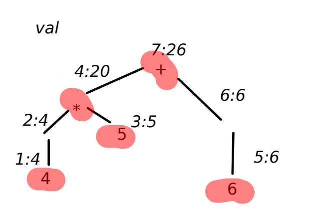


$S \rightarrow E$ =	{ display(E.val); }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val); }
| T	{ E.val = T.val; }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val); }
| F	{ T.val = F.val; }
$F \rightarrow$ (E)	{ F.val = E.val; }
| num	{ F.val = num.val; }

Do attribute evaluation bottom-up.
Evaluate the following expression:

4 * 5 + 6

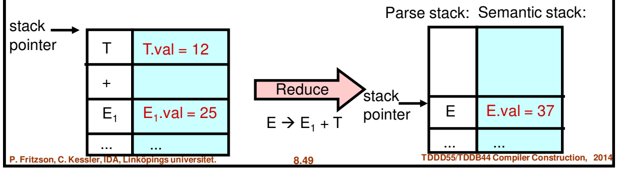
Small Attribute Grammar Exercise



LR Implementation of Attribute Grammars

- **In an 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

      $A.attr = f(\beta_1.attr, \ldots, \beta_k.attr)$



LR Implementation of Attribute Grammars

- **In an LR parser (comment to picture on the previous slide)**
  - A semantic action: **E.val = E1.val +T.val**
    translated to
    a statement: **val[stkp-2] = val[stkp-2]+val[stkp]**
  - Comments:
    - stkp denotes the stack pointer, val the attribute value (an array)
    - its value in the semantic action is the value *before* the reduction
    - Af the call, the LR parser will reduce stkp by the length of the right hand side of grammar rule (here: 3)
    - It then puts E on the parse stack (because we reduced with $E = E_1+T$) with the result that the stack pointer increases a step and we get the reduced configuration in the previous slide.

LR Implementation of Attribute Grammars

Generated semantic routine:

```
semantic(ruleno)
{ switch ruleno
case 1: display(val[stkp-1]);
case 2: val [stkp-2] = val [stkp-2] + val[stkp];
case 3: ;
case 4: val[stkp-2] = val[stkp-2] * val[stkp];
case 5: ;
case 6: val[stkp-2] = val[stkp-1];
case 7: val[stkp] = num.val;
}
```

Grammar:

1.	$S \rightarrow E =$
2.	$E \rightarrow E_1 + T$
3.	| T
4.	$T \rightarrow T_1 * F$
5.	| F
6.	$F \rightarrow (E)$
7.	| num

- stkp specifies the stack pointer before reducing
- The stack grows with higher addresses
- reduce pops with stkp := stkp – lengthRightHandSide(rule)

Implementation of Attribute Grammars

- **In a Recursive Descent Parser:**
  - Recall: One procedure for each nonterminal
  - **Interpretation**:
    - Add a *formal parameter* for each attribute
      - implicit semantic stack (i.e., by *parameters* stored on the normal program execution stack)
      - parameters for synthesized attributes to be passed by reference, so values can be returned
  - **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 (EBNF style):

$S \rightarrow E =$	{ display(E.val); }
$E \rightarrow T_1$	{ $E.val = T_1.val$; }
$\{+ T_2\}$	{ $E.val = E.val + T_2.val$; }
$T \rightarrow F_1$	{ $T.val = F_1.val$; }
$\{* F_2\}$	{ $T.val = E.val * F_2.val$; }
$F \rightarrow (E)$	{ F.val = E.val; }
| 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 = *E_val +T2_val;
  }
}
```