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.

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 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.

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.

Semantic analysis and intermediate code generation
The method used in this phase is syntax-directed translation.

Aim 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.

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

generates code for the transformation:

```pascal
a := IntToReal(b);
```

`IntToReal` is a function for changing integers to a floating-point value.

Aim 2: Intermediate code generation

Another representation of the source code is generated.

Generation of intermediate code has, among others, the following advantages:

- machine-independent
- not profiled for a certain language
- suitable for optimization
- can be used for interpreting

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

(Also called reverse Polish notation)

Example:

<table>
<thead>
<tr>
<th>Infix</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>a b +</td>
</tr>
<tr>
<td>a + b * c</td>
<td>a b c * +</td>
</tr>
<tr>
<td>(a + b) * c</td>
<td>a b c *</td>
</tr>
<tr>
<td>a + (-b - 3 * c)</td>
<td>a b 3 c * - +</td>
</tr>
</tbody>
</table>

where @ denotes unary minus.

- 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....)
Given an arithmetic expression in reverse Polish 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: Evaluate the postfix expression below.

\[ a \ b \ @ \ 3 \ c \ * \ - \ + \]

Given that \( a = 34, \ b = 4, \ c = 5 \)

corresponding infix notation: \( a + (-b - 3 \times c) \)

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>abcl@+-</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>bc@+-</td>
</tr>
<tr>
<td>3</td>
<td>34 4</td>
<td>@+-</td>
</tr>
<tr>
<td>4</td>
<td>34 -4</td>
<td>@+</td>
</tr>
<tr>
<td>5</td>
<td>34 -4 3</td>
<td>c*+</td>
</tr>
<tr>
<td>6</td>
<td>34 -4 3 5</td>
<td>*+</td>
</tr>
<tr>
<td>7</td>
<td>34 -4 15</td>
<td>+</td>
</tr>
<tr>
<td>8</td>
<td>34 -19</td>
<td>+</td>
</tr>
<tr>
<td>9</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Extending Polish notation

- Assignment
  - \( := \) binary operator,
  - lowest priority for infix form,
  - uses the l-value for its first operand

Example:

\[ x := 10 + k \times 30 \]

\[ x \ 10 \ k \ 30 \ * \ + \ := \]

- Conditional statements
  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.

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:

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:
Remember that:

\[
\text{while } <\text{expr}> \text{ do } <\text{stat}>
\]

gives us

L2: \(<\text{expr}>\ L1 \text{ JEQZ } <\text{stat}>\ L2 \text{ JUMP } L1:\)

Exercise:

Translate the repeat and for statements to postfix notation.

Suitable data-structure

An array where label corresponds to index.

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

Correspond to a reduced variant of parse trees. A parse tree contains redundant information, see the figure below.

Example: Parse trees for \(a := b \times c + d\)

![Abstract syntax tree for \(a := b \times c + d\):](image)

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 traversing
+ postorder traversing gives us postfix notation!
- Far from machine code

Implementation of AST

The tree is flattened, suitable for external storage.

![Implementation of AST](image)
Three-address code

\[
\begin{array}{c}
\text{addr}_1 \quad \text{op} \quad \text{addr} _2 \\
\end{array}
\]

\[
\begin{array}{c}
X := X \quad \text{op} \quad Y \\
\end{array}
\]

\[
\begin{array}{c}
\text{addr} _1 \quad \text{addr} _2 \quad \text{addr} _3 \\
\end{array}
\]

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

Quadruples

Form:

\[
\begin{array}{c}
\text{op} \quad \text{arg} _1 \quad \text{arg} _2 \quad \text{res} \\
\end{array}
\]

Example: Assignment statement \(A := B * C + D\)
gives us the quadruples

\[
\begin{array}{c}
T_1 := B * C \\
T_2 := T_1 + D \\
A := T_2 \\
\end{array}
\]

\[
\begin{array}{c}
\text{op} \quad \text{arg} _1 \quad \text{arg} _2 \quad \text{res} \\
\end{array}
\]

\[
\begin{array}{c}
* \quad B \quad C \quad T_1 \\
+ \quad T_1 \quad D \quad T_2 \\
:= \quad T_2 \quad A \\
\end{array}
\]

\(T_1, T_2\) are temporary variables.
The contents of the table are references to the symbol table.

Control structures using quadruples

Example:

\[
\begin{align*}
\text{if } a &= b \\
\text{then } x &= x + 1 \\
\text{else } y &= 20;
\end{align*}
\]

\[
\begin{array}{c}
\text{Quad-no} \quad \text{op} \quad \text{arg} _1 \quad \text{arg} _2 \quad \text{res} \\
\end{array}
\]

\[
\begin{array}{c}
1 \quad = \quad a \quad b \quad T1 \\
2 \quad \text{JEQZ} \quad T1 \quad (6) \dagger \\
3 \quad + \quad x \quad 1 \quad T2 \\
4 \quad := \quad T2 \quad x \\
5 \quad \text{JUMP} \quad (7) \dagger \\
6 \quad := \quad 20 \quad y \\
7 \quad \\
\end{array}
\]

† The jump address was filled in later as we can not know in advance the jump address during generation of the quadruple in a pass.
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.

Array-reference

\(A[I] := B\)

\[
\begin{array}{c}
\text{op} \quad \text{arg} _1 \quad \text{arg} _2 \quad \text{res} \\
\end{array}
\]

\[
\begin{array}{c}
[ ]= \quad A \quad I \quad T1 \\
:= \quad B \quad T1 \\
\end{array}
\]

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

\(B := A[I]\)

\[
\begin{array}{c}
\text{op} \quad \text{arg} _1 \quad \text{arg} _2 \quad \text{res} \\
\end{array}
\]

\[
\begin{array}{c}
[ ]= \quad A \quad I \quad T2 \\
:= \quad T2 \quad B \\
\end{array}
\]

\([ ]=\) is called r-value, specifies the value of an element
**Triples** (also called two-address code)

Form:

```
<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>+</td>
<td>(1)</td>
<td>D</td>
</tr>
<tr>
<td>:=</td>
<td>A</td>
<td>(2)</td>
</tr>
</tbody>
</table>
```

Example: \( A := B \times C + D \)

No temporary name!

**Quadruples vs triples**

**Quadruples:**
- Temporary variables take up space in the symbol table.
- Good control over temporary variables.
- Easier to optimise 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**

There are two 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. **Translation scheme**
   - Describe the translation process using
     - a) a CFG
     - b) a number of semantic operations (without attributes)
     - \( A \rightarrow XYZ \) {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**

**Semantic analysis**

Intuition: Attach semantic actions to syntactic rules to perform semantic analysis and intermediate code generation.

The example below describes part of a CFG for variable declarations in a small language. Assume that the source language contains 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> \rightarrow ... \\
<decl> \rightarrow var <name-list> : <type-id> 
{Attach the type of <type-id> to all id in <name-list>} \\
<name-list> \rightarrow <name-list>, <name> 
{Check that name in <name-list> is not duplicated, and check that name has not been declared previously} \\
<name-list> \rightarrow <name> 
{Check that name has not been declared previously} \\
<type-id> \rightarrow "ident" 
{Check in the symbol table for "ident", return its index if it is already there, otherwise error: unknown type.} \\
<name> \rightarrow "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.

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( E \rightarrow E_1 + T ) {print(‘+’)}</td>
<td></td>
</tr>
<tr>
<td>2 (</td>
<td>T )</td>
</tr>
<tr>
<td>3 ( T \rightarrow T_1 * F ) {print(‘*’)}</td>
<td></td>
</tr>
<tr>
<td>4 (</td>
<td>F )</td>
</tr>
<tr>
<td>5 ( F \rightarrow ( E ) )</td>
<td></td>
</tr>
<tr>
<td>6 (</td>
<td>id ) {print(id)}</td>
</tr>
</tbody>
</table>

Translation of the input string:

\( a + b * d \) becomes in postfix:

\( a \ b \ d \ * \ + \)

See the parse tree on the next page:
Implementation in the LR case

The parser routine:

```pascal
procedure parser;
begin
  while not done do
  begin
    case action of
    shift:
    ... 
    reduce:
      call semantic(rule);
      ... 
    end (* case *);
  end (* while *);
end (* parser *);
```

procedure semantic(rule);
begin
  case rule of
  1 : write('+');
  3 : write('*');
  6 : write(id);
  end;
end;
```

Attribute grammar

- A way to extend a CFG.
- Each nonterminal will have one or more attributes (value fields).
- A number of semantic rules which calculate the values of the attributes using other attributes.

Attributes can be:

- **Inherited attributes** which are transferred from left to right in a production (and downwards in a parse tree). Examples: type info, addresses for variables.

- **Synthesised attributes** which are transferred from right to left in a production (and upwards in a parse tree). Examples: value of variables, translation to internal form.

Example 1: Attribute grammar

Semantic Analysis

Example of a semantic tree for the string `in+3*r` according to grammar E.

```
E → num
E → num . num
E → id
E → E1 op E2
```

```
```
```
A syntax-directed definition for type checking expressions in the CFG above.

\[
E \rightarrow \text{num} \quad \{E\.type := \text{integer}\}
\]

\[
E \rightarrow \text{num} \cdot \text{num} \quad \{E\.type := \text{real}\}
\]

\[
E \rightarrow \text{id} \quad \{E\.type := \text{lookup-typ(id.entry)}\}
\]

\[
E \rightarrow E_1 \text{ op } E_2 \quad \{E\.type := \begin{cases} \text{if } (E_1\.type = \text{integer}) \\
\text{and } (E_2\.type = \text{integer}) \\
\text{then integer} \\
\text{else if } (E_1\.type = \text{integer}) \\
\text{and } (E_2\.type = \text{real}) \\
\text{then real} \\
\text{else if } (E_1\.type = \text{real}) \\
\text{and } (E_2\.type = \text{integer}) \\
\text{then real} \\
\text{else if } (E_1\.type = \text{real}) \\
\text{and } (E_2\.type = \text{real}) \\
\text{then real} \\
\text{else error} \end{cases} \}
\]

Example of a syntax-directed definition for type conversion during intermediate code generation. Details are shown as comments to make the example readable.

\[
E \rightarrow E_1 \text{ op } E_2 \\
S \rightarrow V := E \\
\{ \text{if } V\.type = E\.type \text{ then ...} \text{ (* generate code directly according to type*)} \}
\{ \text{else if } (V\.type = \text{integer}) \text{ and } (E\.type = \text{real}) \text{ then ...} \text{ (* Semantic error: TYPE ERROR! *)} \}
\{ \text{else if } (V\.type = \text{real}) \text{ and } (E\.type = \text{integer}) \text{ then ...} \text{ (* Code generation with type conversion: } E\.value := ...; V\.value := \text{IntToReal}(E\.value) \text{ *)} \}
\}
\]

Example 2: Attribute grammar

Intermediate code generation

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E \rightarrow E_1 + T) { E.Code := \text{E1.Code }</td>
<td></td>
</tr>
<tr>
<td>(E_1 \cdot T) { E.Code := E_1.Code</td>
<td></td>
</tr>
<tr>
<td>(T) { E.Code := \text{T.Code} }</td>
<td></td>
</tr>
<tr>
<td>(T \rightarrow '0') { T.Code := '0' }</td>
<td></td>
</tr>
<tr>
<td>('1') { T.Code := '1' }</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>('9') { T.Code := '9' }</td>
<td></td>
</tr>
</tbody>
</table>

Code is an attribute which is attached to all nonterminals in the grammar.

There is a semantic rule for each grammar rule attached to the left hand side, which calculates the value of the attribute Code (the code produced) just for this nonterminal.

Example 3: Attribute grammar

Calculator: Interpreting in a bottom-up environment

See the example below of a calculator, i.e. an interpreter for arithmetic expressions, which calculates the value of an arithmetic expression, without generating any intermediate code.

Each nonterminal has a synthesised attribute val.

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S \rightarrow E =) { \text{display}(E.val) }</td>
<td></td>
</tr>
<tr>
<td>(E \rightarrow E_1 + T) { E.val := E_1.val + T.val }</td>
<td></td>
</tr>
<tr>
<td>(T) { E.val := T.val }</td>
<td></td>
</tr>
<tr>
<td>(T \rightarrow T_1 * F) { T.val := T_1.val * F.val }</td>
<td></td>
</tr>
<tr>
<td>(F) { T.val := F.val }</td>
<td></td>
</tr>
<tr>
<td>(F \rightarrow (E)) { F.val := E.val }</td>
<td></td>
</tr>
<tr>
<td>(\text{Int} \rightarrow \text{Int}, \text{digit}) { \text{Int.val} := \text{Int.val} + \text{lexval} }</td>
<td></td>
</tr>
<tr>
<td>(\text{digit}) { \text{Int.val} := \text{lexval} }</td>
<td></td>
</tr>
</tbody>
</table>

Input: \(25 + 4 \times 3 = \)
Input: $25 + 4 \times 3 = $

How can we make a program from this?

Observations:

- Here all attributes are synthesised.
- I rule no. 2, "+" denotes a symbol in the production and the addition operation in the semantic rule.
- `$\text{lexval}$` is the value of a number character which returns from the scanner in the form: `<digit, lexval>`
  i.e. the number character is converted to a corresponding integer in the scanner.

Implementation in the LR case

To be able to propagate attributes we introduce a semantic stack which grows in parallel with the parse stack (same stack pointer is used).

When we are ready to perform a reduction the semantic action will synthesise a new attribute whose value is a function of the attributes belonging to the symbols of the right side.

That is, if the production is

$$A \rightarrow \alpha$$

As attributes $b$ are calculated by the formula

$$b := f(c_1, c_2, ..., c_k)$$

where $c_1, c_2, ..., c_k$ are the attributes belonging to the symbols in $\alpha$.

Example: when we are about to perform the reduction $E \rightarrow E_1 + T$ the stack pointer points to $T$.

```
<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Semantic stack</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>stkp</td>
<td>T.val</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>E_1</td>
<td>E_1.val</td>
<td>-2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

We perform the semantic action

$$E.val := E_1.val + T.val$$

with the statement

```
```

Comments:

- `$\text{stkps}$` denotes the stack pointer.
- Its value in the semantic action above is before the reduction.
• After the call the LR parser will reduce \( \text{stk}_p \) by the length of the right side (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 following configuration:

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Semantic stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( E.\text{val} )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

\( \text{stk}_p \)

procedure semantic(rule);
begin
  case rule of
  1: write(val[stk_p-1]);
  2: val[stk_p-2]:= val[stk_p-2]+val[stk_p];
  3: ;
  4: val[stk_p-2]:= val[stk_p-2]*val[stk_p];
  5: ;
  6: val[stk_p-2]:= val[stk_p-1];
  7: ;
  8: val[stk_p-1]:= val[stk_p-1]*10+lexval
  9: val[stk_p]:= lexval
  end;
end;

\( (\text{lexval} \) is a global variable from the scanner)

NB!
• \( \text{stk}_p \) specifies the stack pointer before reducing.
• The stack grows with higher addresses.
• reduce pops with
  \[ \text{stk}_p := \text{stk}_p - |\beta| \]
  at the reduction \( A \rightarrow \beta \)

Example 4: Attribute grammar

Calculator: Interpreting in the recursive descent case

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. ( S \rightarrow E )</td>
<td>{write(( E.\text{val} ))}</td>
</tr>
<tr>
<td>1. ( E \rightarrow T_1 )</td>
<td>{( E.\text{val} := T_1.\text{val} )}</td>
</tr>
<tr>
<td></td>
<td>{(+T_2)} {( E.\text{val} := T_1.\text{val} + T_2.\text{val} )}</td>
</tr>
<tr>
<td>2. ( T \rightarrow F_1 )</td>
<td>{( T.\text{val} := F_1.\text{val} )}</td>
</tr>
<tr>
<td></td>
<td>{(*F_2)} {( T.\text{val} := F_1.\text{val} \times F_2.\text{val} )}</td>
</tr>
<tr>
<td>3. ( F \rightarrow (E) )</td>
<td>{( F.\text{val} := E.\text{val} )}</td>
</tr>
<tr>
<td>4</td>
<td>{(</td>
</tr>
</tbody>
</table>

Implementation: Add a parameter for each attribute.

procedure \( E(\text{var } e_{\text{val}} : \text{integer})\);
var \( t_{\text{val}} : \text{integer} \);
begin
  \( T(t_{\text{val}}) \);
  \( e_{\text{val}} := t_{\text{val}} \);
  while (token = '+'\) do begin
    scan;
    \( T(t_{\text{val}}) \);
    \( e_{\text{val}} := e_{\text{val}} + t_{\text{val}} \);
  end;
end;

Synthesised attributes become Var parameters since they return Values.
Syntax-directed generation of quadruples for assignment statements and arithmetic expressions

Bottom-up analysis

1. ass \rightarrow var := E
2. E \rightarrow E_1 + T
3. \mid T
4. T \rightarrow T_1 * F
5. \mid F
6. F \rightarrow ( E )
7. \mid id
8. var \rightarrow id

Attribute:

\text{adr} \quad \text{address/index in symbol table}

Functions:

\text{GEN}(op, arg_1, arg_2, res) \quad \text{generates quadruple}

\text{GENTEMP()} \quad \text{generates new temp-variable and returns address to it}

\text{LOOKUP(id)} \quad \text{returns the address to the identifier}

Syntax-directed translation:

1. \text{GEN}('=', E.adr, _, var.adr);
2. temp := GENTEMP();
   \text{GEN}('+', E_1.adr, T.adr, temp);
   E.adr := temp;
3. E.adr := T.adr;
4. temp := GENTEMP();
   \text{GEN}('*', T_1.adr, F.adr, temp);
   T.adr := temp;
5. T.adr := F.adr;
6. F.adr := E.adr;
7. F.adr := LOOKUP(id);
8. var.adr := LOOKUP(id);

Example of generation of quadruples:

\text{ass} \rightarrow var := E
\text{E} \rightarrow E_1 + T
\mid T
T \rightarrow T_1 * F
\mid F
F \rightarrow ( E )
\mid id
var \rightarrow id

Generating quadruples for typical control structures (replaces sections 8.5 - 8.6 in the book)

IF-statement:

IF <E> THEN <S>₁ ELSE <S>₂

Quadruples for the above statement generally appear as:

\text{in: quadruples for Temp := <E>}
\text{p: JEQF Temp q+1} \quad \text{Jump over <S>₁ if <E> false}
\text{quadruples for <S>₁}
\text{q: JUMP r} \quad \text{Jump over <S>₂}
\text{q+1: quadruples for <S>₂}
\text{r: \ldots}

To be able to put in the jumps we want, the grammar is factorised to:

1. <if-stat> ::= <true-part> <S>₂
2. <true-part>::= <if-clause> <S>₁ ELSE
3. <if-clause>::= IF <E> THEN

Attributes:

\text{ADDR} = \text{address to the symbol table for the result of <E>}
\text{QUAD} = \text{quadruple number}

Functions:

\text{NEXTQUAD} = \text{produces next quadruple number.}
\text{GEN} = \text{creates and fills in a quadruple.}

Datastructure:

Generated quadruples are stored in a matrix:

\text{QUADR}[1..N, 1..4] \quad \text{of quads}
Syntax directed translation scheme with attributes “Attribute grammar” for translating the IF statement

3. <if-clause> ::= IF <E> THEN
   
   {<if-clause>.QUAD := NEXTQUAD;
   Save the address to the next quadruple, i.e. the one that generates jump over <S>_2.

   GEN(JEQF, <E>.ADDR, 0, 0)
   Jump to <S>_2. Location q+1 not yet known!
   }

2. <true-part> ::= <if-clause> <S>_1 ELSE
   
   {<true-part>.QUAD := NEXTQUAD;
   Save next quadruple number, i.e. the one which generates jump over <S>_2.

   GEN(JUMP, 0, 0, 0);
   Jump to next statement. Location r not yet known.
   QUADR[<if-clause>.QUAD, 3] := NEXTQUAD
   Insert quadruple number given by (backpatch) <if-clause>.QUAD at position q+1.
   }

1. <if-stat> ::= <true-part> <S>_2
   
   {QUADR[<true-part>.QUAD, 2] := NEXTQUAD}
   Done. Fix the jump to the next stmt, (backpatch) i.e. to location r in <true-part>.QUAD.
   }

   (A real attribute grammar does not have side effects such as GEN)

WHILE-statement: WHILE <E> DO <S>

Quadruples for the statement above generally appear as :

   in: quadruples for Temp := <E>  
   p: JEQF 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 to the end of <S> }

Exercise:

Show how quadruples can be generated for the REPEAT-UNTIL statement.