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.

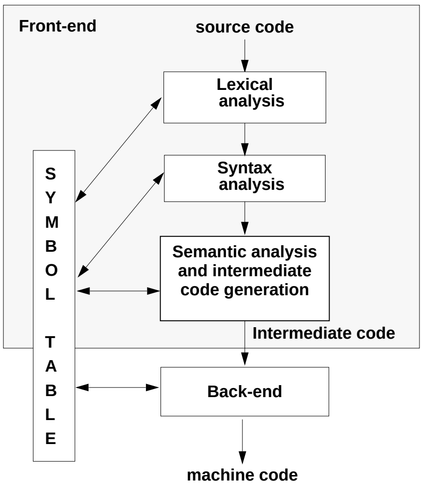
Follow-up questions:

- 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


Front-end

source code

Lexical analysis

Syntax analysis

Semantic analysis and intermediate code generation

Intermediate code

Back-end

machine code

SYMBOL TABLE

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.

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

generates code for the transformation:

```
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:

The internal form is:

+ 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:

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 * - +`

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 * 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 notation

- **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 * + :=
  ```

- **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`.

  `L1 LABEL` (eller `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:

```
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:

```
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

An *array* where label corresponds to index.

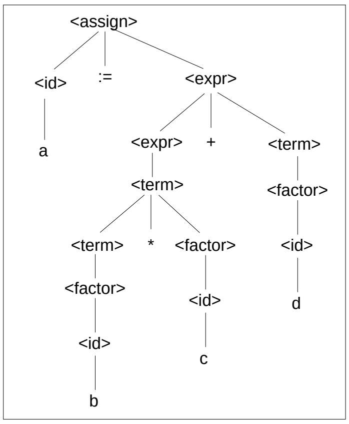
Elements:

- Operand
  Pointer to the symbol table.
- Operator
  A numeric code, for example, which does not collide with the symbol table index.

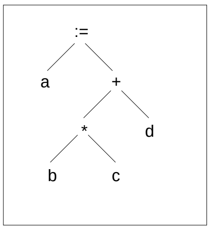
Abstract syntax trees

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

Example: Parse trees for `a := b * c + d`



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

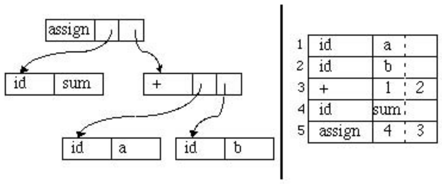


Advantages and disadvantages of abstract syntax trees

- \+ 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.



1	id	a	
2	id	b	
3	+	1	2
4	id	sum	
5	assign	4	3

Three-address code

```
z   :=   x   op   y
↑        ↑        ↑

addr1    addr2    addr3
```

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

Quadruples

Form:

op	***arg$_1$***	***arg$_2$***	***res***

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

gives us the quadruples

```
T1 :=  B * C
T2 :=  T1 + D
A  :=  T2
```

op	arg$_1$	arg$_2$	res
*	B	C	T1
+	T1	D	T2
:=	T2		A

`T1`, `T2` are temporary variables.

The contents of the table are references to the symbol table.

Control structures using quadruples

Example:

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

Quad-no	op	arg$_1$	arg$_2$	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 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**.

Procedure call

Example:f(a$_1$, a$_2$, ..., a$_n$)

op	arg$_1$	arg$_2$	res
param	a$_1$		
param	a$_2$		
...	...		
param	a$_n$		
call	f	n	

Example: `READ(X)`

op	arg$_1$	arg$_2$	res
param	X		
call	READ	1	

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

op	arg$_1$	arg$_2$	res
*	A	B	T1
+	X	5	T2
param	T1		
param	T2		
call	WRITE	2	

Array-reference

`A[I] := B`

op	arg$_1$	arg$_2$	res
[]=	A	I	T1
:=	B		T1

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

`B := A[I]`

op	arg$_1$	arg$_2$	res
=[]	A	I	T2
:=	T2		B

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

Triples (also called **two-address code**)

Form:

$op \quad arg_1 \quad arg_2$

Example: `A := B * C + D`

	op	arg_1	arg_2
1	*	B	C
2	+	(1)	D
3	:=	A	(2)

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 → 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> → ...
<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.

	Productions	Semantic operations
1	$E \rightarrow E_1$ `+ T`	`{print('+')}`
2	`| T`	
3	$T \rightarrow T_1$ `* F`	`{print('*')}`
4	`| F`	
5	`F → ( E )`	
6	`| id`	`{print(id)}`

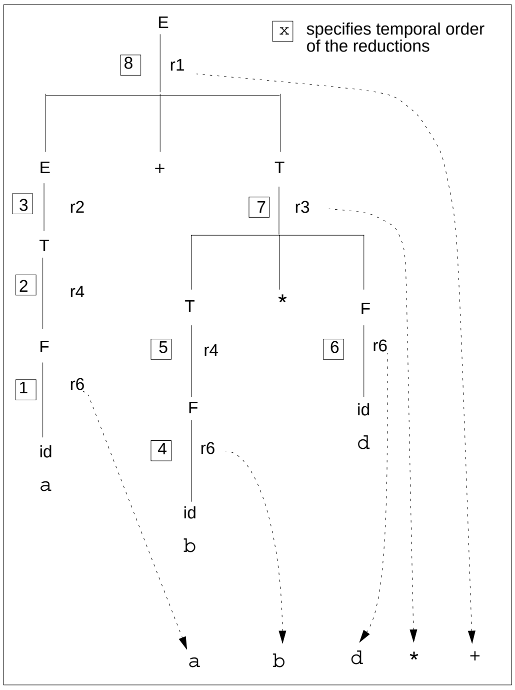
Translation of the input string:

`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:

```
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 Analys

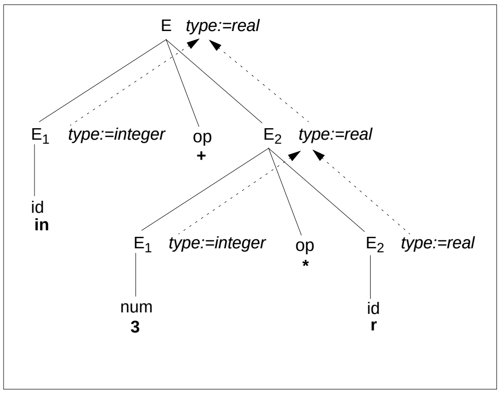
Example of a semantic tree for the string `in+3*r` according to grammar E.

$E \rightarrow \text{num}$

$E \rightarrow \text{num . num}$

$E \rightarrow \text{id}$

$E \rightarrow E_1 \text{ op } E_2$



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

```
E → num          {E.type := integer}
E → num . num    {E.type := real}
E → id           {E.type:=  lookup-typ(id.entry)}
E → $E_1$ op $E_2$     { E.type :=   if   ($E_1$.type = integer)
                                     and  ($E_2$.type = integer)
                                then integer
                                else if  ($E_1$.type = integer)
                                         and ($E_2$.type = real)
                                     then real
                                else if  ($E_1$.type = real)
                                     and ($E_2$.type = integer)
                                     then real
                                else if  ($E_1$.type = real)
                                         and ($E_2$.type = real)
                                         then real
                                else error }
```

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

```
E → $E_1$ op $E_2$
S  → V := E
    { if V.type = E.type
    then ...  (* generate code directly according to type*)
       else if (V.type = integer) and (E.type= real)
            then ...  (* Semantic error: TYPE ERROR! *)
             else if  (V.type = real)
                      and (E.type= integer)
                  then ...(* Code generation with
                             type conversion:

                            E.value:= ...;

                            V.value:= IntToReal(E.value)
                         *)
     }
```

Example 2: Attribute grammar

Intermediate code generation

Translating expressions in the language over G(E) to reverse Polish notation.

Productions	Semantic rules
E → E_1 + T	{ E.*Code*:= E1.*Code* || T.*Code* || '+' }
| E_1 - T	{ E.*Code* := E_1.*Code* || T.*Code* || '-' }
| T	{ E.*Code* := T.*Code* }
T → '0'	{ T.*Code* := '0' }
| '1'	{ T.*Code* := '1' }
...	...
| '9'	{ T.*Code* := '9' }

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

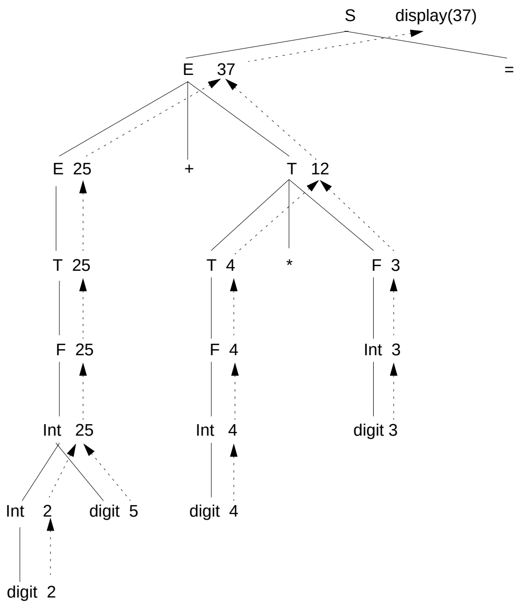
See the example below of a calculator, i.e. an interpreter for arithmetic expressions, which caluclates the value of an arithmetic expression, without generating any intermediate code.

Each nonterminal has a synthesised attribut `val`.

	Productions	Semantic operations
1	S → E =	{ display(E.val) }
2	E → E_1 + T	{ E.val := E_1.val + T.val }
3	| T	{ E.val := T.val }
4	T → T_1 * F	{ T.val := T_1.val * F.val }
5	| F	{ T.val := F.val }
6	F → (E)	{ F.val := E.val }
7	| Int	{ F.val := Int.val }
8	Int → Int_1 digit	{ Int.val := Int_1.val*10 + *lexval* }
9	| digit	{ Int.val := *lexval* }

Input: `25 + 4 * 3 =`

Input: `25 + 4 * 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.
- *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

$A \rightarrow \alpha$

A:s 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 α.

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

	Parse stack	Semantic stack	Position
`stkp`	`T`	`T.val`	`0`
	`+`		`-1`
	E_1	E_1`.val`	`-2`
	...	...	...

We perform the semantic action

`E.val := E`$_1$`.val + T.val`

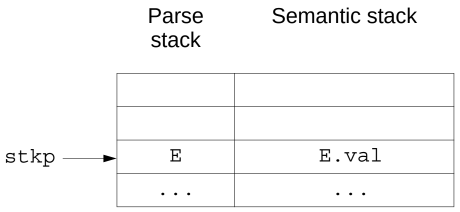
with the statement

```
val[stkp-2]:= val[stkp-2]+ val[stkp];
```

Comments:

- `stkp` denotes the stack pointer.
- Its value in the semantic action above is _*before*_ the reduction.

- After the call the LR parser will reduce `stkp` 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:



---

compare with semantic actions on page 171

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

(*lexval* is a global variable from the scanner)

NB!

- `stkp` specifies the stack pointer **before** reducing.
- The stack grows with higher addresses.
- *reduce* pops with

  `stkp := stkp - |β|`

  at the reduction A → β

Implementation in the case of recursive descent

- Interpretation

  When it is a matter of pure parsing we have a procedure for each nonterminal. Add a parameter for each attribute - this can be regarded as an *implicit stack.*

- Code generation

  Write the translated code to a file.

Example 4: Attribute grammar

Calculator: Interpreting in the recursive descent case

	Productions	Semantic operations
0.	S → E =	{`write(E.val)`}
1.	E → T_1	{E.val := T_1.val}
	{ + T_2 }	{E.val := T_1.val + T_2.val}
2.	T → F_1	{T.val := F_1.val}
	{ * F_2}	{T.val := F_1.val * F_2.val}
3.	F → (E)	{F.val := E.val}
4	| **integer**	{F.val := lexval}

Implementation: Add a parameter for each attribute.

```
procedure E(var e_val : integer);
var  t_val : integer;
begin
   T(t_val);
   e_val := t_val;
   while (token = '+') do
   begin
      scan;
      T(t_val);
      e_val := e_val + t_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. | T
4. T $\rightarrow$ T_1 * F
5. | F
6. F $\rightarrow$ (E)
7. | id
8. var $\rightarrow$ id

Attribute:

`adr`	address / index in symbol table

Functions:

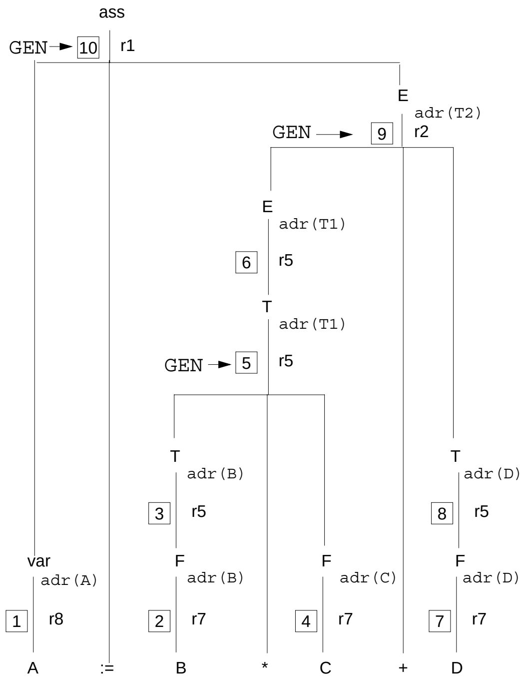
`GEN(op,arg1,arg2,res)`	generates quadruple
`GENTEMP()`	generates new temp-variable and returns address to it
`LOOKUP(id)`	returns the address to the identifier

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

Syntax-directed translation:

1. GEN(':=', E.adr, _ , var.adr);
2. temp := GENTEMP();
   GEN('+', E_1.adr, T.adr, temp);
   E.adr := temp;
3. E.adr := T.adr;
4. temp := GENTEMP();
   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:



Generated quadruples for input: `A := B * C + D`

at $\boxed{5}$ `GEN('*',adr(B),adr(C),adr(T1))`
at $\boxed{9}$ `GEN('+',adr(T1),adr(D),adr(T2)`
at $\boxed{10}$ `GEN(':=',adr(T2), _ ,adr(A))`

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:

in: quadruples for Temp := <E>
p: JEQF Temp q+1 Jump over $<S>_1$ if <E> false
quadruples for $<S>_1$
q: JUMP r Jump over $<S>_2$
q+1: quadruples for $<S>_2$
r: ...

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:

`ADDR` = address to the symbol table for the result of <E>.
`QUAD` = quadruple number

Functions:

`NEXTQUAD` = produces next quadruple number.
`GEN` = creates and fills in a quadruple.

Datastructure:

Generated quadruples are stored in a matrix:
`QUADR[1..N, 1..4] (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>`$_1$.

`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.