Semantic Analysis and Intermediate Code Generation

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.
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.

```plaintext
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:
  - + 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
  - Triples
- **Infix notation**
  - Example:
    ```plaintext
    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)
  - Examples and comparison:
    ```plaintext
    Infix   Postfix
    a + b   a b +
    (a + b) * c   a b + c *
    a + (-b - 3 * c)   a @ 3 b - c * +
    ```
- 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 ...)

**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.
Extending Polish/Postfix Notation

Assignment Statement

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

Example:

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

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

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

\[
\text{if } a+b \text{ then } \\
\text{if } c-d \text{ then } \\
\text{else } y := 20 \\
\text{else } z := 30;
\]

gives us

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

Representing While

Suitable Data Structure for Postfix Code

\[
\text{while } <expr> \ do \ <stat> \\
\text{gives us} \\
L2: <expr> L1 JEQZ <stat> L2 JUMP L1;
\]

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.

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.

\[
L1 \ \text{LABEL} \ {or} \ L1: \\
<\text{label}> \ \text{JUMP} \\
<\text{value}> <\text{label}> \ \text{JEQZ} \\
(\text{value} = 0 \ \Rightarrow \ \text{false}, \ \text{otherwise} \ \Rightarrow \ \text{true})
\]

Example 1:

\[
\text{IF } <\text{expr}> \ \text{THEN } <\text{statement1}> \ \text{ELSE } <\text{statement2}> \\
gives us \\
<\text{expr}> L1 \ JEQZ <\text{statement1}> L2 \ JUMP L1: <\text{statement2}> L2: \\
\text{where } L1: \ \text{stands for } L1 \ \text{LABEL}
\]

Small Postfix Notation Exercise

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

\[
a := b \cdot c + d
\]

Abstract syntax tree for

\[
a := b \cdot c + d
\]
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 traversal gives us postfix notation!
  - Far from machine code

Three-address Code and Quadruples

Three-address code:

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

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.
  - Temporary variables take up space in the symbol table.
  - Easier to optimise and move code around.

Triples (also called two-address code)

Triples Form:

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

Triples:

- Temporary variables take up space in the symbol table.
- Good control over temporary variables.
- Easier to optimise and move code around.
- 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:

- 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.

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.

Example 2: Translation Schema Intermediate Code Generation

The parser routine:

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

The semantic routine:

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

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

```
E1 + T   {print('+')}
T1 * F   {print('*')}
```

Produces

```
1 E -> E1 + T  (print('+'))
2 T -> T1 + F  (print('+'))
3 T1 -> T   . . .
4 F -> F   . . .
5 F1 -> (E)  . . .
6 id -> id (print(id))
```

Translate the input string:
```
a + b * d
```

Becomes in postfix:
```
a b d * +
```

See the parse tree on the coming page:

Parse Tree of Translation to Postfix Code

```
E1 + T   {print('+')}
T1 * F   {print('*')}
```

This method is a more procedural form of the previous one (contains implementation details), which explicitly show the evaluation order of semantic rules.

2. Syntax Directed Translation Scheme

Describe the translation process using:

- a) a CFG
- b) a number of semantic operations
  - e.g. a rule: \( A \rightarrow XYZ \) (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.
Syntax-directed translation of assignment statements and arithmetic expressions into quadruples

using a bottom-up approach

### Generating Quadruples

<table>
<thead>
<tr>
<th>Quadruple</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. S → Var := E</td>
<td>{ GEN (ASGN, E.adr, 0, Var.adr); }</td>
</tr>
<tr>
<td>2. E → E₁ + T</td>
<td>{ temp = gen_tempvar(); GEN (ADD, E₁.adr, T.adr, temp); E.adr = temp; }</td>
</tr>
<tr>
<td>3.</td>
<td>T</td>
</tr>
<tr>
<td>4. T → T₁ * F</td>
<td>{ temp = gen_tempvar(); GEN (MUL, T₁.adr, F.adr, temp); T.adr = temp; }</td>
</tr>
<tr>
<td>5.</td>
<td>F</td>
</tr>
<tr>
<td>6. F → ( E )</td>
<td>{ F.adr = E.adr; }</td>
</tr>
<tr>
<td>7.</td>
<td>id</td>
</tr>
<tr>
<td>8. Var → id</td>
<td>{ Var.adr = lookup(id.name); }</td>
</tr>
</tbody>
</table>

### Generating Quadruples for Control Structures

**Example: IF-THEN-ELSE**

- S → if E then S₁ else S₂
- Jump to S₂ if E is false/zero
- After S₁ jump to after S₂

**Problem:** jump target quadruple indices q₁, r are unknown when the jumps are generated

**Solution:** factorise the grammar, store jump index in attribute `quad`

**Generate Quadruples for if-then-else (2)**

1. `<ifstmt>` ::= `<truepart>` S₂
2. `<truepart>` ::= `<ifclause>` S₁ 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 q₁ not known yet.  
   \}

2. `<truepart>` ::= `<ifclause>` S₁ else
   
   \{ <truepart>.quad = currentquad + 1;  
   // save address q of jump over S₂ for later  
   GEN (JUMP, 0, 0, 0);  
   // jump over S₁. Target r not known yet.  
   QUADRUPLE[<ifclause>.quad][2] = currentquad + 1;  
   // backpatch JEQZ target to q₁  
   \}

3. `<ifstmt>` ::= `<truepart>` S₂
Generate Quadruples for if-then-else (4)

1. <ifstmt> ::= <truepart> S2
   \{ QUADRUPLE[<truepart>.quad][1] = currentquad + 1;
   // backpatch JUMP target to (r-1)+1 \}

2. <truepart> ::= <ifclause> S1
   \{ <truepart>.quad = currentquad + 1;
   // save address q of jump over S2 for later
   GEN(JUMP, 0, 0, 0);
   // j S
   \}

3. <ifclause> ::= if E then …

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>}

Similarly: while statement, repeat statement …

Small Quadruple Generation Exercise

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
q: JUMP in Jump to the loop-predicate q1:

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>}

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 1+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 = Inttype)
E → E1 op E2 (E.type = Inttype)
E → E1 op E2 (E.type = Realtype)
E → id (E.type = Inttype)
E → num (E.type = Inttype)
E → lookup(id.name).type (E.type = Inttype)
E → E1 op E2 (E.type = ... (see next page))

Attribute grammar for syntax-directed type checking

\[
\begin{align*}
E &\rightarrow \text{num} \quad \{ \ E.type = \text{Inttype}; \} \\
E &\rightarrow \text{num} \ . \ \text{num} \quad \{ \ E.type = \text{Realtype}; \} \\
E &\rightarrow \text{id} \quad \{ E.type = \text{tokup.id\_name\_type}; \} \\
E &\rightarrow E_1 \ \text{op} \ E_2 \quad \{ E.\text{t\_type} = \text{lookup}(E_1.\text{name})\_\text{t\_type}; \} \\
&\quad \{ E.type = (E_1.\text{type} == \text{Inttype}) && (E_2.\text{type} == \text{Inttype}) ? \text{Inttype} : (E_1.\text{type} == \text{Inttype}) && (E_2.\text{type} == \text{Realtype}) \lor (E_1.\text{type} == \text{Realtype}) && (E_2.\text{type} == \text{Inttype}) \lor (E_1.\text{type} == \text{Realtype}) && (E_2.\text{type} == \text{Realtype}) \? \text{Realtype} : \text{error(\text{Type error})}, \text{Notype}; \}
\end{align*}
\]


Attribute grammar extended for assignment statement with implicit type conversion from integer to Real

\[
\begin{align*}
S &\rightarrow \text{V := E} \quad \{ E.\text{type} = \ldots ; \} \\
S &\rightarrow \text{V := E} \quad \text{if } (V.\text{type} == E.\text{type}) \text{ generate code directly according to type} \\
&\quad \text{else if } (V.\text{type} == \text{Inttype}) && (E.\text{type} == \text{Realtype}) \text{ error(\text{Type error})}; \\
&\quad \text{else if } (V.\text{type} == \text{Realtype}) && (E.\text{type} == \text{Inttype}) \\
&\quad \text{\text{Code generation / evaluation with type conversion:} E.\text{value} = \ldots ; \}
&\quad \text{V.\text{value} = ConvertIntToReal( E.\text{value} );} \\
&\quad \text{else} \text{\text{Code generation / evaluation with type conversion:} E.\text{value} = \ldots ; \}
&\quad \text{V.\text{value} = ConvertRealToInt( E.\text{value} );} \\
\end{align*}
\]


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

\[
\begin{align*}
E &\rightarrow E_1 \ + \ E_2 \\
&\quad \{ E.\text{code} = \text{concat}( E_1.\text{code}, E_2.\text{code}, \text{"+"}); \} \\
E &\rightarrow E_1 \ - \ T \\
&\quad \{ E.\text{code} = \text{concat}( E_1.\text{code}, T.\text{code}, \text{"-"}); \} \\
T &\rightarrow T_1 \ * \ F \\
&\quad \{ T.\text{code} = \text{"0"}; \} \\
&\quad \{ T.\text{code} = \text{"1"}; \} \\
&\quad \{ T.\text{code} = \ldots ; \} \\
F &\rightarrow ( E ) \\
&\quad \{ F.\text{code} = \text{E.\text{code}}; \} \\
&\quad \{ F.\text{code} = \text{num.\_value}; \} \\
&\quad \{ F.\text{code} = \ldots ; \}
\end{align*}
\]


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

\[
\begin{align*}
S &\rightarrow E = \\
E &\rightarrow E_1 \ + \ T \\
&\quad \{ E.\text{val} = E_1.\text{val} + T.\text{val}; \} \\
&\quad \{ T.\text{val} = T.\text{value}; \} \\
T &\rightarrow T_1 \ * \ F \\
&\quad \{ T.\text{val} = T_1.\text{val} \ * \ F.\text{val}; \} \\
&\quad \{ F.\text{val} = F.\text{value}; \} \\
&\quad \{ F.\text{val} = E.\text{val}; \} \\
&\quad \{ F.\text{val} = \text{num.\_val}; \}
\end{align*}
\]


Small Attribute Grammar Exercise

\[
\text{value of integer-constant token num as computed by the scanner}
\]
**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 \( \text{attr} \) by
      \[
      A.\text{attr} = f ( \beta_1.\text{attr}, \ldots, \beta_k.\text{attr} )
      \]

**LR Implementation of Attribute Grammars**

- In an LR parser (comment to picture on the previous slide)
  - A semantic action: \( E.\text{val} = E1.\text{val} + T.\text{val} \) translated to a statement: \( \text{val}[\text{stk}-2] = \text{val}[\text{stk}-2] + \text{val}[\text{stk}] \)
  - Comments:
    - \( \text{stk} \) denotes the stack pointer, \( \text{val} \) the attribute value (an array)
    - its value in the semantic action is the value before the reduction
    - At the call, the LR parser will reduce \( \text{stk} \) 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 \( E = E1+T \)) with the result that the stack pointer increases a step and we get the reduced configuration in the previous slide.

**Generated semantic routine:**

```c
int semantic(int ruleno) {
  switch(ruleno) {
  case 1: display(val[stk-1]); break;
  case 3: ; break;
  case 5: ; break;
  case 6: val[stk-2] = val[stk-1]; break;
  case 7: val[stk] = num.val; break;
  }
  return 0;
}
```

**Example: Calculator for Recursive Descent**

LL(1) grammar for calculator (EBNF style):

```c
S -> E
E -> T1 (+ T2) E1
T -> F1 (+ F2) F3
F -> ( E ) | num
```

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