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.

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

generates code for the transformation:
```plaintext
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:
    ```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
  - (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
  - `a + b`
  - `a + b * c`
  - `(a + b) * c`
  - `a + (b - 3 * c)`

- Postfix
  - `a b +`
  - `a b c * +`
  - `a b + 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.

Examples Evaluation of Postfix Notation
- Example: evaluate the postfix expression below.
  ```plaintext
  a b @ 3 c * +
  ```

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>1 a b</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>1 a b 3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>1 a b 3 c</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>1 a b 3 c +</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 a b 3 c +</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1 a b 3 c + 5</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>1 a b 3 c + 5 15</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>1 a b 3 c + 5 15</td>
<td>15</td>
</tr>
</tbody>
</table>
Extending Polish/Postfix Notation

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

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

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

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

\[ \text{if } a+b \text{ then} \]
\[ \text{if } c-d \text{ then} \]
\[ \text{else } x := 10 \]
\[ \text{else } y := 20 \]
\[ \text{else } z := 30; \]

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

Small Postfix Notation Exercise

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

Abstract syntax tree for
\[ a := b \times c + d : \]
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

Three-address Code and Quadruples

- Three-address code
  - op: +, -, *, /, :=, JEQZ, JUMP, [ ], =
  - 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.

Control Structures Using Quadruples

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

Array-reference

- A[i] := B
  - [] 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]

Procedure call

- Example: f(a1, a2, ..., an)
  - Quad-no  op  arg1  arg2  res
    1 param a1
    2 param a2
    3 param ...
    n param an
  - Quad-no  op  arg1  arg2  res
    1 call f
    n-1 call i

Quadruples vs triples

- Triples
  - Form:
    - Example: Assignment statement
      A := B * C + D
    - gives us the triples
      T1 := B * C
      T2 := T1 + D
      A := T2
  - Temporary variables take up space in the symbol table.
  - No temporary name!
  - Quad-no  op  arg1  arg2  res
    1 := A B T1
    2 := X 5 T2
    3 param T1
    4 param T2
    5 call WRITE 2

- Quadruples
  - Form:
    - Example: Assignment statement
      A := B * C + D
    - gives us the quadruples
      T1 := B * C
      T2 := T1 + D
      A := T2
  - Temporary variables take up space in the symbol table.
  - Quad-no  op  arg1  arg2  res
    1 := A B
    2 := (1) C

Triples (also called two-address code)

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

- Quadruples:
  - Temporary variables take up space in the symbol table.
  - Quad-no  op  arg1  arg2  res
    1 := A B
    2 := (1) C

- Triples:
  - Know nothing about temporary variables.
  - Take up less space.
  - Optimization by moving code around is difficult, in this case indirect triples are used.

Quad-no  op  arg1  arg2  res
1 := A B
2 := (1) C
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: \( A \rightarrow XYZ \) (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 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.

```plaintext
<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.

Transation of the input string: \( a + b \times d \)
becomes in postfix: \( a \ b \ d \ * \ + \)

See the parse tree on the coming page:

Translation Schema Intermediate Code Generation, Implementation in LR Case

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

The semantic routine:
```c
void semantic(int ruleNo);
{
  switch ruleNo {  // switch ruleNo {
    case 1: print("+");
    case 2: print("*");
    case 3: print("-");
    case 4: print("*");
    case 5: print("+");
  }
}
```

The parse tree of translation to postfix code:

```
```

Productions Semantic operations
1 E → E1 + T (print("+")) 2 T → T1 * F (print("*")) 3 T → T1 + F (print("+")) 4 F → ( E ) (print("(")) 5 F → id (print("id"))
Generating Quadruples

Syntax-directed translation of assignment statements and arithmetic expressions into quadruples using a bottom-up approach

Generating Quadruples

1. \( S \rightarrow \text{Var} \Rightarrow E \)  
   \( \{ \text{GEN ASSIGN, E.adr, 0, Var.adr}; \} \)
2. \( E \Rightarrow E_1 + T \)  
   \( \{ \text{temp = gen_tempvar(); GEN ADD, E1.adr, T.adr, temp; E.adr = temp;} \} \)
3. \( T \Rightarrow T \)  
   \( \{ E.adr = Tadr; \} \)
4. \( T \Rightarrow T_1 * F \)  
   \( \{ \text{temp = gen_tempvar(); GEN MUL, T1 adr, F.adr, temp; T.adr = temp;} \} \)
5. \( F \Rightarrow F \)  
   \( \{ Tadr = Fadr; \} \)
6. \( F \Rightarrow \text{(E)} \)  
   \( \{ Fadr = E.adr; \} \)
7. \( \text{id} \Rightarrow \text{id} \)  
   \( \{ Fadr = lookup(id.name); \} \)
8. \( \text{Var} \Rightarrow \text{id} \)  
   \( \{ \text{Var.adr = lookup(id.name); } \} \)

Generating Quadruples for Control Structures

Example: IF-THEN-ELSE

1. \( S \rightarrow \text{if} \ E \text{ then } S_1 \text{ else } S_2 \)
   - Jump to \( S_2 \) if \( E \) is false/zero
   - After \( S_1 \) jump to after \( S_2 \)

2. Problem: jump target quadruple indices \( q+1, r \) are unknown when the jumps are generated

3. Solution: factorise the grammar, store jump index in attribute \( \text{quad} \)

Generate quadruples for if-then-else (2)

Factorised grammar:

1. \( <\text{ifstmt}> ::= <\text{truepart}> S_2 \)
2. \( <\text{truepart}> ::= <\text{ifclause}> S_1 \text{ else} \)
3. \( <\text{ifclause}> ::= \text{if } E \text{ then} \)

Attributes:

\( \text{addr} \) = address to the symbol table entry for result of \( E \)
\( \text{quad} \) = quadruple number

Generate quadruples for if-then-else (3)

3. \( <\text{ifclause}> ::= \text{if } E \text{ then} \)
   \( \{
   \text{ifclause}.quad = \text{currentquad} + 1;
   // save address p of jump over S1, for later in \text{ifclause}.quad
   \text{GEN(JEQZ, E.adr, 0, 0});
   // jump to S2. Target q+1 not known yet.
   \}
   \)
2. \( <\text{truepart}> ::= <\text{ifclause}> S_1 \text{ else} \)
   \( \{
   \text{truepart}.quad = \text{currentquad} + 1;
   // save address q of jump over S2, for later
   \text{GEN(JUMP, 0, 0, 0, 0});
   // jump over S2. Target r not known yet.
   \text{QUADRUPLE}[<\text{ifclause}>.quad][2] = \text{currentquad} + 1;
   // backpatch JEQZ target to q+1
   \}
   \)
3. \( <\text{ifstmt}> ::= <\text{truepart}> S_2 \)
Generate Quadruples for if-then-else (4)

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

2. `<truepart>` ::= `<ifclause>` `S` else

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

1. `<ifstmt>` ::= `<truepart>` `S`

   `{ 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>`

1. `GEN ( JUMP, `<E>`.ADDR, 0, 0)`
   Jump position not yet known!

2. `{<while-clause>.QUAD := <while>.QUAD;`
   Move along start of `<E>`

   `<while-clause`.NXTQ := NEXTQUAD;
   Save the address to the next quadruple.

3. `{<while-clause>.QUAD := NEXTQUAD;
   QUADRUPLE[ <while-clause>.NXTQ, 3 ] := NEXTQUAD;
   (backpatch) Position at the end of `<S>` }

Small Quadruple Generation Exercise

Attribute Grammars

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)

Grammar rules:

```
E → num     { E.type = Inttype; }
E → num . num { E.type = Realltype; }
E → id      { E.type = lookup(id.name).type; }
E → E1 op E2 { E.type = ... (see next page) }

Semantic rules:

```

Synthesized attribute: type

```
E1.type + E2.type = E.type Realltype
E1.type * E2.type = E.type Realltype
```

Peter Fritzson, Christoph Kessler
IDA, Linköpings universitet, 2014
Attribute grammar for syntax-directed type checking

E → num (E.type = Inttype;)
E → num . num (E.type = Realtype;)
E → id (E.type = lookup(id.name).type;)
E → E1 op E2 { E.type = (E1.type == Inttype && E2.type == Inttype)? Inttype : (E1.type == Inttype && E2.type == Realtype || E1.type == Realtype && E2.type == Inttype) ? Realtype : error("Type error"), Notype; }

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

E → E1 op E2 (E.type = ...)
S → V := E { E.type = ... }
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: 235- 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 → E1 + E2 (E.code = concat(E1.code, E2.code, "+");)
E → E1 - T (E.code = concat(E1.code, T.code, "-");)
T → '0' (T.code = "0");
T → '1' (T.code = "1");
T → ... (T.code = "9");

value of integer constant token num as computed by the scanner

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

Example: Calculator for Recursive Descent

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

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow T_1 \\
E & \rightarrow T_1 + T_2 \\
T_1 & \rightarrow F_1 \\
F_1 & \rightarrow \text{num} \\
F_1 & \rightarrow F_1 * T_2 \\
T_2 & \rightarrow \text{num} \\
T_2 & \rightarrow E_1 \text{val} \\
E_1 & \rightarrow \text{val} \\
E_1 & \rightarrow \text{val} + E_1 \text{val} \\
E_1 & \rightarrow \text{val} * E_1 \text{val} \\
E_1 & \rightarrow \text{val} / E_1 \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
E_1 & \rightarrow \text{val} \text{val} \text{val} \\
\end{align*}
\]

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;
    }
}