Chapter 5

Intermediate Code Generation

- Let us see where we are now.
  - We have tokenized the program and parsed it.
  - We know the structure of the program and of every statement in it,
  - and we have presumably established that it is free of grammatical errors.
  - It would appear that we are ready to start translating it.

1. Semantic Actions and Syntax-Directed Translation

- We can attach a meaning to every production
- Because the sequence of productions guides the generation of intermediate code, we call this process Syntax-Directed Translation.
- **Def:** The computations or other operations attached to the productions impute meaning to each production, and so these operations are called Semantic Actions

- **Def:** The information obtained by the semantic actions is associated with the symbols of the grammar; it is normally put in fields of records associated with the symbols; these fields are called attributes

- **Note:** as far as the parser is concerned, neither the semantic actions nor the attributes are a part of the grammar; they are only used as a device for bridging the gap between parsing and constructing an intermediate representation.

- Things we must take care of with the semantic actions:
  - making sure the variables are declared before use.
  - type checking
  - making sure actual and formal parameters are matched
- These things are called semantic analysis
- So we can now have it both ways, we can Put context dependent information and actions together into a language that is still context free.

2. Intermediate Representations

- We will look at several different representations
  - Syntax Trees
  - Directed Acyclic Graphs
  - Postfix notation
  - Three-Address Code
  - Other Forms.

2.1 Syntax Trees

- typically used when intermediate code is to be generated later (maybe after an optimization pass)

- Statement: x = a * b + a * b
- Parse Tree

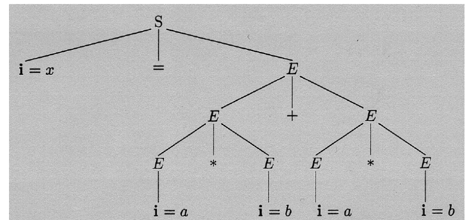

Figure 5.2(a)

- Syntax Tree

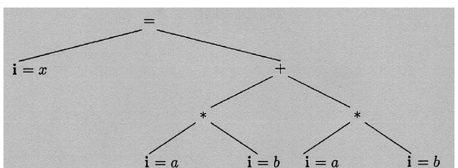

Figure 5.2(b)

- In a Parse Tree the emphasis is on the grammatical structure of the statement.
- In a Syntax Tree the emphasis is on the actual computation to be performed.

2.2 Directed Acyclic Graphs

- The directed acyclic graph (DAG) is a relative of a Syntax Tree.
- The difference is that nodes for variables or repeated sub-expressions are merged.

- DAG -- Notice that this is not the same computations as in the previous examples
  - They had a typo -- (swapped the * and +)
  - But it still explains the concept

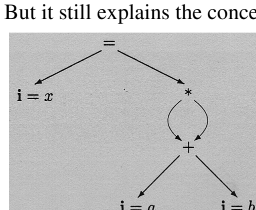

Figure 5.2(c)

- The use of DAG's to eliminate redundant code is our first instance of optimization. We will see more optimization in Chapter 6.
- Redundant code really comes into play when we do array subscripts.
- When you start generating intermediate code, you will be amazed at how much is generated for array subscripts.

2.3 Postfix Notation

- Very easy to generate from a Bottom-Up parse.
- You can also generate it from a Syntax Tree via a postorder traversal.
- The chief virtue of postsfix is that it can be evaluated with the use of a stack.
- Nested if statements can cause problems.

2.4 Three-Address Code

- This form breaks the program down into elementary statements having no more than 3 variables and no more than one operator.
- Sample Statement: x = a + b * b
- Translation:
  - T := b * b
  - x := a + T
- Note: T is a temporary variable.

- The notation is a compromise; it has the general form of a high-level language, but the individual statements are simple enough that they map into assembly language in a reasonably straight forward manner.
- 3AC may be:
  - Generated from a traversal of a Syntax Tree or a DAG.
  - or it may be generated as intermediate code directly in the course of the parse.

2.5 Intermediate Languages

- Sometimes the Intermediate representation may be a language of its own.
- This helps uncouple the front end of the compiler from the back end.
- You can then have a front end for each language that generate the same intermediate language, and then one back end for each type of computer.

- Examples:
  - UNCOL (1961) -- UNiversal Compiler-Oriented Language.
  - P-Code (1981) -- UCSD -- based upon a p-code interpreter (they also built p-code compilers.)
  - GNU Intermediate Code -- gcc, g++, g77, gada,
    - -- a Lispish type intermediate language.

3. Bottom-Up Translation

- Bottom-Up parsing generally lends itself to intermediate code generation more readily than does top-down parsing.
- In either case, we must keep track of the various elements or pieces of the intermediate representation we are using, so we can get at them when we need them.

- These elements will be attributes of symbols in the grammar.
  - For an identifier, the attribute will usually be its address in the symbol table.
  - For a non-terminal, the attribute will be some appropriate reference to part of the intermediate representation.

- The most convenient way to keep track of these attributes is by keeping them in a stack (known as the *semantic stack*).
- In the case of bottom-up parsing, the semantic stack and the parser stack move in synchronism.
  - When we pop from the parse stack we pop the semantic stack, and when we push something onto the parse stack we will push something onto the semantic stack

3.1 Trees

- Syntax Trees
  - The semantic action associated with each production will include planting a tree and taking care of the grafts.
  - The attribute will contain a pointer to the root of the tree for this expression.
  - You will need functions like make_tree( ) and make_leaf( )

- DAGS
  - The main difference between constructing a DAG and constructing a Syntax Tree is that we do not create redundant nodes in a DAG.
  - That means that the functions to create trees and grafts must be modified to check for duplicates.
    - This is typically done with a hash function.

3.2 Postfix Notation

- Postfix notation is particularly easy to generate from a bottom-up parse.
- Grammar:

$S \rightarrow \mathbf{i} = E$	{ output ('=', $\mathbf{i}.lexeme$) }
$E \rightarrow E + E$	{ output ('+') }
$E \rightarrow E * E$	{ output ('*') }
$E \rightarrow (E)$	{ do nothing }
$E \rightarrow \mathbf{i}$	{ output ($\mathbf{i}.lexeme$) }

3.3 Three-Address Code

- We can obtain 3AC from a tree or a DAG, or we can generate it directly in the course of a bottom-up parse.
- To generate 3AC during the parse we need the following functions:
  - MakeQuad( ) -- puts its parameters into the listing file in the proper format.
  - GetTemp( ) -- Generates a new temporary number.

4. Top-Down Translation

- It is, to be blunt, a mess
- Two stacks, one for the parser, and one for the attributes. Thus the semantic stack must now be separate and will not move in synchronism with the parser stack

- Up to now, we picked up attributes from children, and the bottom-up parser provided a handy way to do this. Now we have to be able to transfer attributes between siblings, and in some cases it may be necessary to transfer them from parent to child.

4.1 Synthesized and Inherited Attributes

- synthesized -- from your children (like we have looked at).
- inherited -- from your parent, or sibling
- **Def:** A grammar in which all attributes are synthesized is called an <u>S-attribute grammar</u>

- Notice that inheritance so far is from left to right.
  - That may not always be the case, and that can really mess things up.
- **Def:** Grammars in which no non-terminal ever inherits from a younger brother are called <u>L-attributed grammars</u>

4.2 Attributes in a Top-Down Parse

- Since so many of our problems arise from the need to eliminate left recursion in the underlying grammar, the place to start is to see how a normal semantic action has to be modified when removing left recursion.

- In this example, .s is used to denote a synthesized attribute, and .i an inherited one.

$$
\begin{aligned}
&E \rightarrow T\ \{Q.i := T.s\}\ Q\ \{E.s := Q.s\}\\
&Q_1 \rightarrow +T\ \{Q_2.i := Q_1.i + T.s\}\ Q_2\ \{Q_1.s := Q_2.s\}\\
&Q \rightarrow \epsilon\ \{Q.s := Q.i\}\\
&T \rightarrow F\ \{R.i := F.s\}\ R\ \{T.s := R.s\}\\
&R_1 \rightarrow *F\ \{R_2.i := R_1.i * F.s\}\ R_2\ \{R_1.s := R_2.s\}\\
&R \rightarrow \epsilon\ \{R.s := R.i\}
\end{aligned}
$$

- An sample parse tree of a+b

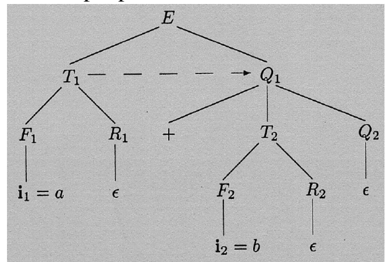

Figure 5.8

4.3 Removal of Left Recursion

- When transforming a grammar to remove left recursions, we must also transform the semantic actions, as the example we just saw implies.
- There are some basic rules on how to do this, but we will not cover them at this time.

5. More about Bottom-Up Translation

- We are now going to consider the cases where semantic actions must be embedded in the midst of the right-hand side of a production.
- And cases where it is expedient to use inherited attributes.

5.1 Embedded Semantic Actions

- You will recall that in S-attributed grammars, the semantic actions occur in one piece at the end of the right-hand side of the production.
- For some statements that is fine, but for others it is not.
  - The typical example is: if C then S
  - because based upon the value of C we need a conditional jump to the end of the statement S.

- There are a few ways around this:
  - Add Semantic actions in the middle of the production (YACC/BISON allow this)
  - Breaking Productions up
  - Adding Marker Non-Terminals
- Another problem:
  - The limit a jump non zero can go (jnz limit). On the Intel architecture, this is 127 bytes.

5.2 Inherited Attributes

- Example: Fortran Variables
  - The data type comes first, so you can enter lexemes into the symbol table with the data type (an inherited attribute)

6. Pascal-Type Declarations

- Problem:
  - The data type comes last, so you can't add the variables into the Symbol table as they come.
- Solution:
  - Save the identifiers (or their symbol table pointer) onto some stack/list
  - When the data types come around, update the list elements.

7. Type Checking and Coercion

- integer operation / float operation
- is it integer_add, float_add,...
  - The problem is that this is an overloaded operator
- It is up to the Semantic Analyzer
  - to determine which operation is desired and to choose the appropriate implementation.
  - And if the user has specified two incompatible operands, it must take some action.


```
if E.type = integer and T.type = integer then
  begin
  get a temporary variable Q for the result (as usual);
  generate 3AC for integer addition of E and T:
     (iadd, E.loc, T.loc, Q.loc)
  end
else if E.type = real and T.type = real then
  begin
  get a temporary variable for the result;
  generate 3AC for floating-point addition of E and T:
     (fadd, E.loc, T.loc, Q.loc)
  end
else if E.type = real and T.type = integer then
  begin
  get a temporary variable Q1 for a floating-point number;
  generate code to convert T to floating-point form and put it into Q1:
     (float, T.loc, —, Q1.loc);
  get a temporary variable Q2 for the result;
  generate 3AC for floating-point addition of E and Q1:
     (fadd, E.loc, Q1.loc, Q2.loc)
  end
else                { other way around }
  begin
  get a temporary variable Q1 for a floating-point number;
  generate code to convert E to floating-point form and put it into Q1:
     (float, E.loc, —, Q1.loc);
  get a temporary variable Q2 for the result;
  generate 3AC for floating-point addition of Q1 and T:
     (fadd, Q1.loc, T.loc, Q2.loc)
  end;
```


8. Summary

- We have seen some of the principal methods for intermediate code generation and some of the principal problems.
- Now we have nearly reached the conclusion of the front end of the compiler.
- Some front-ends also include some optimization, and some interpreters also stop here.