CS3005D Compiler Design

Winter 2024

Lecture #25

Intermediate Representations

Saleena N

CSED NIT Calicut

March 2024

Intermediate Representations

The front end of a compiler generates an Intermediate Representation(IR) of the source program. Some of the common IRs are:

- Abstract Syntax Trees
- Three-address code
- DAG (Directed Acyclic Graph) for expressions
- Static Single-Assignment Form

Abstract Syntax Trees

Abstract Syntax Trees (or simply *syntax trees*)

- Represents the hierarchical syntactic structure of the source program.
- Each construct is represented by an operator node with sub trees corresponding to the semantically meaningful components of the construct.
- Condensed form of *parse tree*[1], with the non terminal nodes either dropped or replaced by operators.

[1] A parse tree is sometimes referred to as concrete syntax tree.

Parse Tree / Abstract Syntax Tree

Grammar:

$S \rightarrow id = E$

$E \rightarrow E + E \mid E * E \mid id \mid num$

Input string: $x = y + 10$

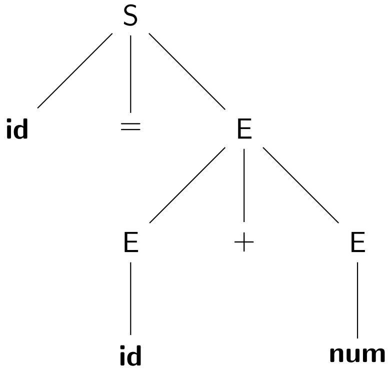


Figure: Parse Tree

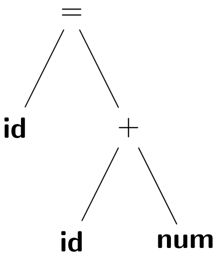


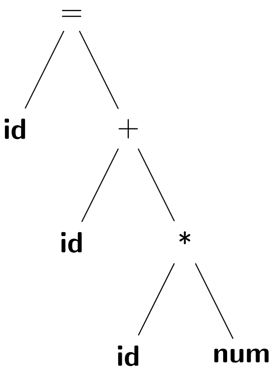
Figure: Abstract Syntax Tree

SDD to construct syntax tree

Production	Semantic Rules
$S \rightarrow \ id = E$	$S.node =$
	$CreateNode('=', \ CreateLeaf(\textbf{id}, \ \textbf{id}.entry), \ E.node)$
$E \rightarrow \ E_1 + E_2$	$E.node = CreateNode('+', \ E_1.node, \ E_2.node)$
$E \rightarrow \ \textbf{num}$	$E.node = CreateLeaf(\textbf{num}, \ \textbf{num}.lexval)$
...	...

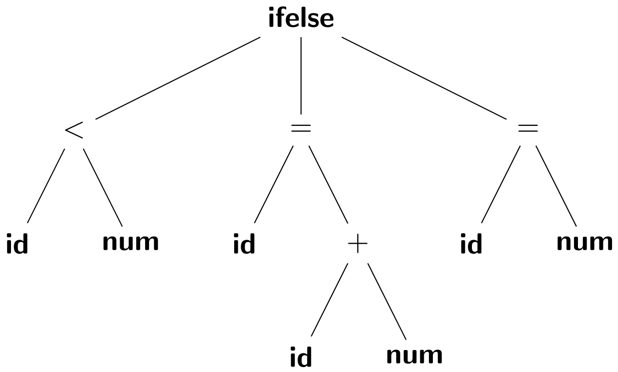
Abstract Syntax Trees

Assignment: $x = sum + i * 10$



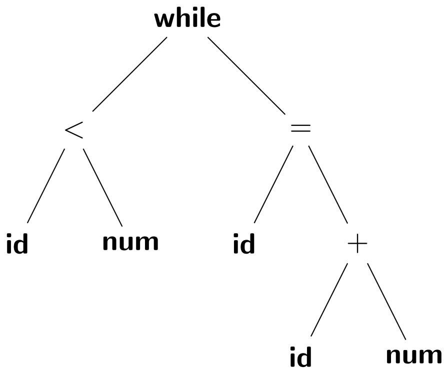
Abstract Syntax Trees

if-else statement: $\textbf{if}(i < 10)\ i = i + 1;\ \textbf{else}\ i = 0;$



Abstract Syntax Trees

while loop: $while(i < 100)\ i = i + 1;$



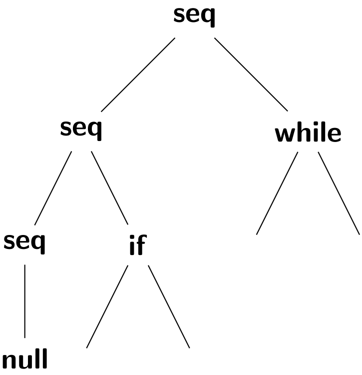
SDD to construct syntax tree

Production	Semantic Rules
$S \rightarrow$ **if** (B) S_1 **else** S_2	$S.node =$ $CreateNode(\textbf{ifelse}, B.node, S_1.node, S_2.node)$
$S \rightarrow$ **while** (B) S_1	$S.node =$ $CreateNode(\textbf{while},\ B.node,\ S_1.node)$
...	...

Note: The nonterminal B denotes a Boolean Expression

Abstract Syntax Trees

Sequence of statements: $slist \rightarrow slist \ stmt \mid \epsilon$
e.g. **if**... followed by **while**...



operator **seq** for a sequence of statements, leaf **null** for an empty statement

Three-Address Code

- Instructions of the form $x = y \; op \; z$ with three addresses - two for the operands and one for the result
- At most one operator on the right side of an instruction
- Temporary names (generated by the compiler) used for intermediate values
- Instructions can have symbolic labels
- Instructions for altering flow of control (conditional/unconditional *goto*s)

Three-Address Code: Addresses

An *address* in a 3-address instruction can be

- a name in the source program
- a constant
- a compiler-generated temporary

e.g. the instruction $t_1 = a + b$ contains 3 addresses: a and b are names in the source program, t_1 is a temporary.
Note: In the implementation, a name is replaced by a pointer to its Symbol Table entry

Three-Address Code: example

The expressions $x = a + b * c$ is translated to the following sequence of 3-address instructions:

$$t_1 = b * c$$
$$t_2 = a + t_1$$
$$x = t_2$$

$t_1, \; t_2$ are compiler generated temporaries

Common 3-address instructions

- Assignment instructions of the form:
  - $x = y \ op \ z$ where *op* is a binary operator and x, y, and z are addresses
  - $x = op \ y$ where *op* is a unary operator and x, y, and z are addresses
- Copy instructions of the form $x = y$
- Unconditional jump `goto` L to transfer control to the instruction labelled L
- Conditional jump
  - `if x goto` L - jump to the instruction labelled L, if x is true
  - `ifFalse x goto` L - jump to the instruction labelled L, if x is false
  - `if x relop y goto` L - jump to the instruction labelled L, if $x \ relop \ y$ is true (*relop* denotes a relational operator)

3-address code: example

$if\ (a < b)\ small = a\ else\ small = b \ldots$

$$
\begin{array}{rl}
 & if\ a < b\ goto\ L_1 \\
 & goto\ L_2 \\
L_1: & small{=}a \\
 & goto\ L_3 \\
L_2: & small{=}b \\
L_3: & \ldots
\end{array}
$$

Common 3-address instructions

- Procedure calls and returns: a procedure call $p(x_1, x_2, \ldots x_n)$ is translated to the following 3-address instructions:

  param x_1
  param x_2
  ...
  param x_n
  call p, n

  Function calls of the form y=call p, n
  Returns of the form return y
- Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. ($x[i]$ refers to the location i memory units beyond x)
- Address and Pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$

3-address code: Representation

- Quardruples - each instruction as a record with four fields: *op* - code for operator, *arg1, arg2* for operands and *result*
- Triples - only three fields for each instruction. Result field is not part of the instruction. An instruction i can use the result of instruction j as operand, by keeping a reference to the position of instruction j
- Indirect Triples - a list of pointers to triples

3-address code: Quardruples

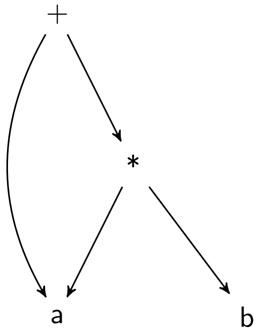
Each instruction as a record with four fields:*op* - code for operator, arg_1, arg_2 for operands and *result*

- Instructions with unary operators do not use arg_2
- *param* uses arg_1 alone
- *goto* instruction keeps target label in *result*

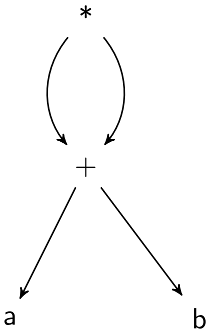
Directed Acyclic Graph(DAG) for Expressions

- Graph nodes corresponding to operands and operators
- Common operand nodes (representing subexpressions) are not replicated
- Can identify *common subexpressions*

Example: DAG for $a + a * b$



Example: DAG for $(a + b) * (a + b)$



$a + b$ is a common subexpression. Code for evaluating $a + b$ needs to be generated only once. Avoids generating redundant code.

Static Single-Assignment (SSA) Form

- All assignments are to variables with distinct names
- SSA facilitates certain code optimizations.
- Definitions of the same variables are changed to definitions of distinct variable by renaming of variables:

$$x = a + b$$
$$\dots$$
$$x = p + q$$

represented in SSA as

$$x_1 = a + b$$
$$\dots$$
$$x_2 = p + q$$

Static Single-Assignment (SSA) Form

SSA: Definition of the same variable in two different control-flow paths - use of ϕ functions:

$if\ (\ldots)\ x = 1\ else\ x = 2;$
$y = x;$

represented in SSA as

$if\ (\ldots)\ x_1 = 1\ else\ x_2 = 2;$
$x_3 = \phi(x_1,\ x_2);$
$y = x_3;$

The ϕ function returns the value of the argument corresponding to the control flow path taken to reach the statment containing it.

References

References:

- Aho A.V., Lam M.S., Sethi R., and Ullman J.D. Compilers: Principles, Techniques, and Tools (ALSU). Pearson Education, 2007.

Further reading:

- ALSU Chapter 2, Chapter 6