CS6013 - Modern Compilers: Theory and Practise

Semantic Analysis

V. Krishna Nandivada

IIT Madras

Acknowledgement

Semantic Processing

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines:

- interpret meaning of the program based on its syntactic structure
- two purposes:
  - finish analysis by deriving context-sensitive information (e.g. type checking)
  - begin synthesis by generating the IR or target code
- associated with individual productions of a context free grammar or subtrees of a syntax tree

Alternatives for semantic processing

- one-pass analysis and synthesis
- one-pass compiler plus peephole
- one-pass analysis & IR synthesis + code generation pass
- multipass analysis (e.g. gcc)
- multipass synthesis (e.g. gcc)
- language-independent and retargetable (e.g. gcc) compilers

Our focus in the assignments: One-pass analysis & IR synthesis + multipass analysis + multipass synthesis.

Type checking

- We need generate type information.
  - For fields, variables, expressions, functions.
- Need to enforce types:
  - Assignments, function calls, expressions.
- We need to remember the type information and recall them as/where required – symbol table.

Symbol tables

For compile-time efficiency, compilers use a symbol table:

- associates lexical names (symbols) with their attributes

What items should be entered?

- variable names
- defined constants
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries (we'll get there)

A symbol table is a compile-time structure
Separate table for structure layouts (types) (includes field offsets and lengths)
May need to preserve list of locals for the debugger

Symbol table information

What kind of information might the compiler need?

- textual name
- data type
- dimension information (for aggregates)
- declaring procedure
- lexical level of declaration
- storage class (base address)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions
- ...

Storage classes of variables

During code generation, each variable is assigned an address (addressing method), approrpriate to its storage class.

- A local variable is not assigned a fixed machine address (or relative to the base of a module) – rather a stack location that is accessed by an offest from a register whose value does not point to the same location, each time the procedure is invoked. Why is it interesting?
- Four major storage classes: global, stack, stack static, registers

Symbol table organization

How should the table be organized?

- Linear List
  - **O**(n) probes per lookup
  - easy to expand — no fixed size
  - one allocation per insertion
- Ordered Linear List
  - **O**$(\log_2 n)$ probes per lookup using binary search
  - insertion is expensive (to reorganize list)
- Binary Tree
  - **O**(n) probes per lookup — unbalanced
  - **O**$(\log_2 n)$ probes per lookup — balanced
  - easy to expand — no fixed size
  - one allocation per insertion
- Hash Table
  - **O**(1) probes per lookup — on average
  - expansion costs vary with specific scheme

Nested scopes: block-structured symbol tables

What information is needed?

- when asking about a name, want most recent declaration
- declaration may be from current scope or outer scope
- innermost scope overrides outer scope declarations

Key point: new declarations occur only in current scope

What operations do we need?

- `void put (Symbol key, Object value)`
  bind key to value
- `Object get(Symbol key)`
  return value bound to key
- `void beginScope()`
  remember current state of table
- `void endScope()`
  close current scope and restore table to state at most recent open beginScope

Nested scopes: complications

Fields and records:

give each record type its own symbol table

or assign record numbers to qualify field names in table

with R **do** ⟨stmt⟩:

- all IDs in ⟨stmt⟩ are treated first as R.id
- separate record tables:
  chain R's scope ahead of outer scopes
- record numbers:
  open new scope, copy entries with R's record number
  or chain record numbers: search using these first

Nested scopes: complications (cont.)

Implicit declarations:

- labels:
  declare and define name (in Pascal accessible only within enclosing scope)
- Ada/Modula-3/Tiger FOR loop:
  loop index has type of range specifier

Overloading:

- link alternatives (check no clashes), choose based on context

Forward references:

- bind symbol only after all possible definitions $\Rightarrow$ multiple passes

Other complications:

packages, modules, interfaces — IMPORT, EXPORT

Attribute information

Attributes are internal representation of declarations
Symbol table associates names with attributes
Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

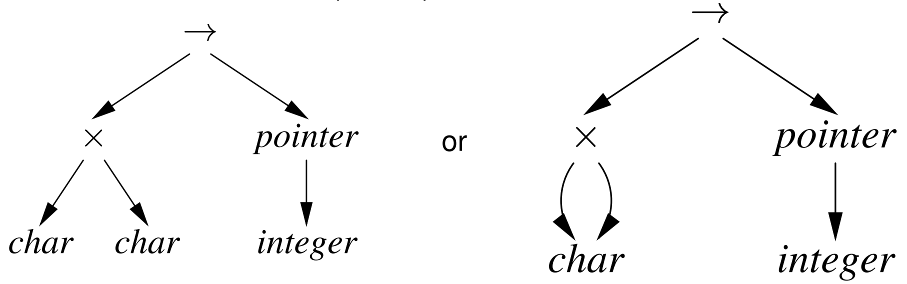
Type expressions

Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, $integer$, $real$, etc.
2. type names
3. constructed types (constructors applied to type expressions):
    1. $array(I,T)$ denotes an array of T indexed over I
       e.g., $array(1 \ldots 10, integer)$
    2. products: $T_1 \times T_2$ denotes Cartesian product of type expressions T_1 and T_2
    3. records: fields have names
       e.g., $record((\mathrm{a} \times integer), (\mathrm{b} \times real))$
    4. pointers: $pointer(T)$ denotes the type "pointer to an object of type T"
    5. functions: $D \to R$ denotes the type of a function mapping domain type D to range type R
       e.g., $integer \times integer \to integer$

Type descriptors

Type descriptors are compile-time structures representing type expressions
e.g., $char \times char \to pointer(integer)$



Type compatibility

Type checking needs to determine type equivalence
Two approaches:

Name equivalence: each type name is a distinct type

Structural equivalence: two types are equivalent iff. they have the same structure (after substituting type expressions for type names)

- $s \equiv t$ iff. s and t are the same basic types
- $array(s_1, s_2) \equiv array(t_1, t_2)$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $s_1 \times s_2 \equiv t_1 \times t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $pointer(s) \equiv pointer(t)$ iff. $s \equiv t$
- $s_1 \to s_2 \equiv t_1 \to t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$

Type compatibility: example

Consider:

```
    type  link  =  ↑cell;
    var   next  :  link;
          last  :  link;
          p     :  ↑cell;
          q, r  :  ↑cell;
```

Under name equivalence:

- `next` and `last` have the same type
- `p`, `q` and `r` have the same type
- `p` and `next` have different type

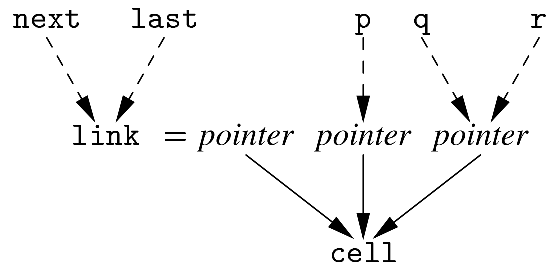
Under structural equivalence all variables have the same type

Ada/Pascal/Modula-2/Tiger are somewhat confusing: they treat distinct type definitions as distinct types, so `p` has different type from `q` and `r`

Type compatibility: Pascal name equivalence

Build compile-time structure called a type graph:

- each constructor or basic type creates a node
- each name creates a leaf (associated with the type's descriptor)



Type expressions are equivalent if they are represented by the same node in the graph

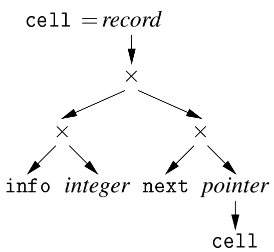
Type compatibility: recursive types

Consider:

```
    type  link  =  ↑cell;
          cell  =  record
                   info  :  integer;
                   next  :  link;
                   end;
```

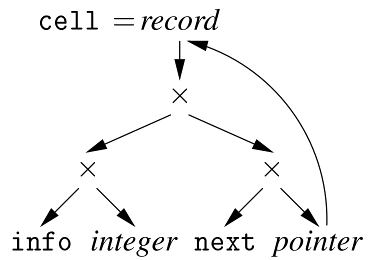
We may want to eliminate the names from the type graph

Eliminating name `link` from type graph for record:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:



Food for thought - fun assignment

Write a Type Checker for BuritoJava expressions.

Considerations:

- Overloaded addition operation.
- Assignment op.
- Function calls.
- Inheritance.

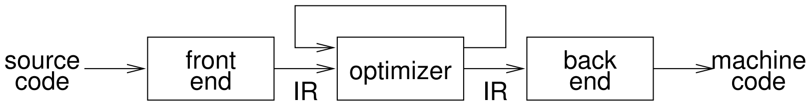
Intermediate representations

Why use an intermediate representation?

1. break the compiler into manageable pieces
   – good software engineering technique
2. simplifies retargeting to new host
   – isolates back end from front end
3. simplifies handling of "poly-architecture" problem
   – m lang's, n targets $\Rightarrow m+n$ components (myth)
4. enables machine-independent optimization
   – general techniques, multiple passes

An intermediate representation is a compile-time data structure

Intermediate representations



Generally speaking:

- front end produces IR
- optimizer transforms that representation into an equivalent program that may run more efficiently
- back end transforms IR into native code for the target machine

Intermediate representations

Representations talked about in the literature include:

- abstract syntax trees (AST)
- linear (operator) form of tree
- directed acyclic graphs (DAG)
- control flow graphs
- program dependence graphs
- static single assignment form
- 3-address code
- hybrid combinations

Intermediate representations - properties

Important IR Properties

- ease of generation
- ease of manipulation
- cost of manipulation
- level of abstraction
- freedom of expression
- size of typical procedure

Subtle design decisions in the IR have far reaching effects on the speed and effectiveness of the compiler.
Level of exposed detail is a crucial consideration.

IR design issues

- Is the chosen IR appropriate for the (analysis/ optimization/ transformation) passes under consideration?
- What is the IR level: close to language/machine.
- Multiple IRs in a compiler: for example, High, Medium and Low

```
x = a[i,j+2]        t1 = j + 2        r1 = [fp-4]
                    t2 = i * 2        r2 = r1 + 2
                    t3 = t1 + t2      r3 = [fp-8]
                    t4 = 4 * t3       r4 = r3 * 20
                    t5 = addr a       r5 = r4 + r2
                    t6 = t5 + t4      r6 = 4 * r5
                    x = *t6           r7 = fp - 216
                                      f1 = [r7+r6]
```

- In reality, the variables etc are also only pointers to other data structures.

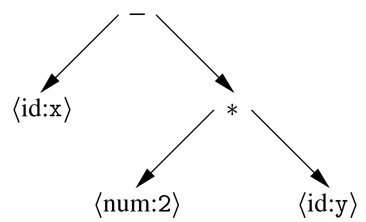
Intermediate representations

Broadly speaking, IRs fall into three categories:

- Structural
  - structural IRs are graphically oriented
  - examples include trees, DAGs
  - heavily used in source to source translators
  - nodes, edges tend to be large
- Linear
  - pseudo-code for some abstract machine
  - large variation in level of abstraction
  - simple, compact data structures
  - easier to rearrange
- Hybrids
  - combination of graphs and linear code
  - attempt to take best of each
  - e.g., control-flow graphs
  - Example: GCC Tree IR.

Abstract syntax tree

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.

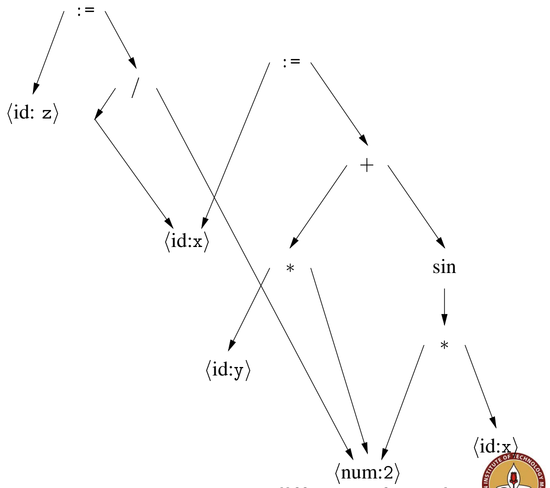


This represents "$x - 2 * y$".
For ease of manipulation, can use a linearized (operator) form of the tree.
e.g., in postfix form: $x\ 2\ y * -$

Directed acyclic graph

A directed acyclic graph (DAG) is an AST with a unique node for each value.

```
x := 2 * y + sin(2*x)
z := x / 2
```



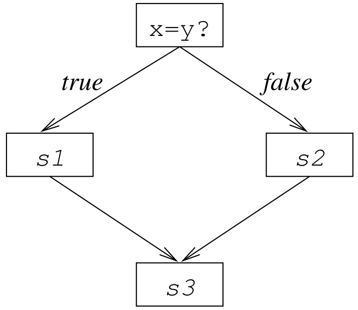
Q: What to do for matching names present across different functions?

Control flow graph

The control flow graph (CFG) models the transfers of control in the procedure

- nodes in the graph are basic blocks
  straight-line blocks of code
- edges in the graph represent control flow
  loops, if-then-else, case, goto

```
if (x=y) then
   s1
else
   s2
s3
```



3-address code

- At most one operator on the right side of an instruction.
- 3-address code can mean a variety of representations.
- In general, it allow statements of the form:
  `x ← y op z`
  with a single operator and, at most, three names.
  Simpler form of expression:
  `x - 2 * y`
  becomes
  ```
  t1 ← 2 * y
  t2 ← x - t1
  ```

Advantages

- compact form (direct naming)
- names for intermediate values

Can include forms of prefix or postfix code

3-address code: Addresses

Three-address code is built from two concepts: addresses and instructions.

- An address can be
  - A name: source variable program name or pointer to the Symbol Table name.
  - A constant: Constants in the program.
  - Compiler generated temporary:

3-address code

Typical instructions types include:

1. assignments $x \leftarrow y\ \underline{op}\ z$
2. assignments $x \leftarrow \underline{op}\ y$
3. assignments $x \leftarrow y[i]$
4. assignments $x \leftarrow y$
5. branches `goto L`
6. conditional branches
   `if x goto L`
7. procedure calls
   param x_1, param x_2, ...param x_n
   and
   `call p, n`
8. address and pointer assignments

How to translate:

```
if (x < y) S1 else
S2
```

?

3-address code - implementation

Quadruples

- Has four fields: op, arg1, arg2 and result.
- Some instructions (e.g. unary minus) do not use arg2.
- For copy statement : the operator itself is =; for others it is implied.
- Instructions like `param` don't use neither arg2 nor result.
- Jumps put the target label in result.

`x - 2 * y`

	op	result	arg1	arg2
(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

- simple record structure with four fields
- easy to reorder
- explicit names

3-address code - implementation

Triples

`x - 2 * y`

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

- use table index as implicit name
- require only three fields in record
- harder to reorder

3-address code - implementation

Indirect Triples

`x - 2 * y`

	exec-order	stmt	op	arg1	arg2
(1)	(100)	(100)	load	y	
(2)	(101)	(101)	loadi	2	
(3)	(102)	(102)	mult	(100)	(101)
(4)	(103)	(103)	load	x	
(5)	(104)	(104)	sub	(103)	(102)

- simplifies moving statements (change the execution order)
- more space than triples
- implicit name space management

Indirect triples advantage

```
for i:=1 to 10 do
begin
 a=b*c
 d=i*3
end
      (a)
```

Optimized version

```
a=b*c
for i:=1 to 10 do
begin
 d=i*3
end
      (b)
```

```
(1)  := 1 i
(2)  * b c
(3)  := (2) a
(4)  * 3 i
(5)  := (4) d
(6)  + 1 i
(7)  LE I 10
(8)  IFT go (2)
```

Execution Order (a) : 12345678
Execution Order (b) : 23145678
Note: No need to change the operands.
Labels still need changing.

Other hybrids

An attempt to get the best of both worlds.

- graphs where they work
- linear codes where it pays off

Unfortunately, there appears to be little agreement about where to use each kind of IR to best advantage.
For example:

- PCC and FORTRAN 77 directly emit assembly code for control flow, but build and pass around expression trees for expressions.
- Many people have tried using a control flow graph with low-level, three address code for each basic block.

Intermediate representations

But, this isn't the whole story
Symbol table:

- identifiers, procedures
- size, type, location
- lexical nesting depth

Constant table:

- representation, type
- storage class, offset(s)

Storage map:

- storage layout
- overlap information
- (virtual) register assignments

Advice

- Many kinds of IR are used in practice.
- There is no widespread agreement on this subject.
- A compiler may need several different IRs
- Choose IR with right level of detail
- Keep manipulation costs in mind

Opening remarks

What have we done so far?

- Compiler overview.
- Scanning and parsing.
- JavaCC, visitors and JTB
- Semantic Analysis - specification, execution, attribute grammars.
- Type checking, Intermediate Representation.

Announcement:

- Assignment 2. Seven days to go.

Today:

- Intermediate code generation.

Gap between HLL and IR

Gap between HLL and IR

- High level languages may allow complexities that are not allowed in IR (such as expressions with multiple operators).
- High level languages have many syntactic constructs, not present in the IR (such as if-then-else or loops)

Challenges in translation:

- Deep nesting of constructs.
- Recursive grammars.
- We need a systematic approach to IR generation.

Goal:

- A HLL to IR translator.
- Input: A program in HLL.
- Output: A program in IR (may be an AST or program text)

Translating expressions

```
S -> id = E;   {gen(top.get(id.lexeme) '=' E.addr);}

E -> E1 + E2   {E.addr = new Temp();
                gen(E.addr '=' E1.addr '+' E2.addr);}

    | - E1     {E.addr = new Temp();
                gen(E.addr '=' - E2.addr);}

    | (E1)     {E.addr = E1.addr;}

    | id       {E.addr = top.get(id.lexeme);}
```

- Builds the three-address code for an assignment statement.
- addr is an `synthetic`-attribute of E.
  - denotes the address that will hold the value of E.
- Constructs a three-address instruction and appends the instruction to the sequence of instructions.
- *top* is the top-most (current) symbol table.

Array elements dereference (Recall)

- Elements are typically stored in a block of consecutive locations.
- If the width of each array element is w, then the i^{th} element of array A (say, starting at the address $base$), begins at the location: $base + i \times w$.
- For multi-dimensions, beginning address of $A[i_1][i_2]$ is calculated by the formula:
  $base + i_1 \times w_1 + i_2 \times w_2$
  where, w_1 is the width of the row, and w_2 is the width of one element.
- We declare arrays by the number of elements (n_j is the size of the j^{th} dimension) and the width of each element in an array is fixed (say w).
  The location for $A[i_1][i_2]$ is given by
  $base + i_1 \times n_2 \times w + i_2 \times w$
- Q: If the array index does not start at '0', then ?
- Q: What if the data is stored in column-major form?

Translation of Array references

- Extending the expression grammar with arrays:

```
S -> id = E;

   | L = E;

E -> E1 + E2

   | id

   | L
```

```
L -> id [E]

   | L1 [E]
```


Translation of Array references (contd)

```
S -> id = E;   {gen(top.get(id.lexeme) '=' E.addr)}

   | L = E;    {gen(L.array.base'['L.addr']' '=' E.addr);}

E -> E1 + E2   {E.addr = new Temp();
                gen(E.addr '=' E1.addr '+' E2.addr);}

   | id        {E.addr = top.get(id.lexeme);}

   | L         {E.addr = new Temp();
                gen(E.addr '=' L.array.base'['L.addr']');}
```

Nonterminal L has three synthesized attributes

1. $L.addr$ denotes a temporary that is used while computing the offset for the array reference.
2. $L.array$ is a pointer to the ST entry for the array name. The field $base$ gives the actual l-value of the array reference.

Translation of Array references (contd)

```
L -> Id [E]  {L.array = top.get(id.lexeme);
              L.type = L.array.type.elem;
              L.addr = new Temp();
              gen(L.addr '=' E.addr'*'L.type.width);}

   | L1 [E]  {L.array = L1.array;
              L.type = L1.type.elem;
              t = new Temp();
              L.addr = new Temp();
              gen(t '=' E.addr '*' L.type.width);
              gen (L.addr '=' L1.addr '+' t);}
```

3. $L.type$ is the type of the subarray generated by L.
   - For any type t: $t.width$ gives get the width of the type.
   - For any type t: $t.elem$ gives the element type.

Translation of Array references (contd)

Example:

- Let a denotes a 2×3 integer array.
- Type of a is given by $array(2, array(3, integer))$
- Width of a = 24 (size of $integer$ = 4).
- Type of $a[i]$ is $array(3, integer)$, width = 12.
- Type of $a[i][j] = integer$

Exercise:

- Write three adddress code for $c + a[i][j]$

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [t3]
t5 = c + t4
```

Q: What if we did not know the size of $integer$ (machine dependent)?

IR generation for flow-of-control statements

```
P->S            S.next = new Label();
                P.code = S.code || label(S.next)

S->assgin       S.code = assign.code

S->if (B) S1    B.true = new Label();
                B.false = S1.next = S.next
                S.code = B.code || label(B.true) || S1.code

S->if (B) S1    B.true = new Label();
   else S2      B.false = new Label();
                S1.next = S2.next = S.next
                S.code = B.code || label(B.true) || S1.code
                         || gen ('goto' S.next)
                         || label (B.false) || S2.code
```

- *code* is an synthetic attribute: giving the code for that node.
- Assume: *gen* only creates an instruction.
- || concatenates the code.

IR generation for flow-of-control statements

```
S->while(B)S1   begin = new Label();
                B.true = new Label();
                B.false = S.next
                S1.next = begin
                S.code = label(S.next) || B.code
                         || label(B.true) || S1.code
                         || gen('goto' begin)

S->S1 S2        S1.next = new Label()
                S2.next = S.next
                S.code = S1.code || label(S1.next) || S2.code
```

- *code* is an synthetic attribute: giving the code for that node.
- Assume: *gen* only creates an instruction.
- || concatenates the code.

IR generation for boolean expressions

```
B -> B1 || B2       B1.true = B.true
                    B1.false = new Label()
                    B2.true = B. true
                    B2.false = B.false
                    B.code = B1.code || label(B1.false) || B2.code

B -> B1 && B2       B1.true = new Label()
                    B1.false = B.false
                    B2.true = B. true
                    B2.false = B.false
                    B.code = B1.code || label(B1.true) || B2.code

B -> !B1            B1.true = B.false
                    B1.false = B.true
                    B.code = B1.code

B -> E1 rel E2      t = new Temp()
                    B.code=E1.code||E2.code||gen(t'='E1.addr rel.op E2.addr)
                              || gen('if' t 'goto' B.true)
                              || gen('goto' B.false);

B -> true           B.code = gen('goto' B.true)

B -> false          B.code = gen('goto' B.false)
```


Some challenges/questions

- Avoiding redundant gotos. ??
- Multiple passes. ??
- How to translate implicit branches: `break` and `continue`?
- How to translate `switch` statements efficiently?
- How to translate procedure code?

Closing remarks

What have we done today?

- Intermediate Code Generation.

To read

- Dragon Book. Sections 6.4, 6.5, 6.6, 6.7, 6.8, 6.9 and 2.8