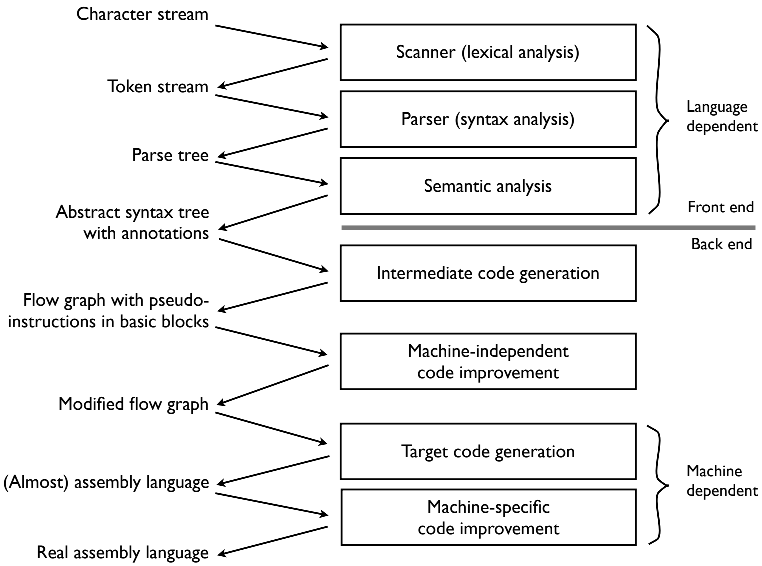
Building a runnable program

Chapter 14

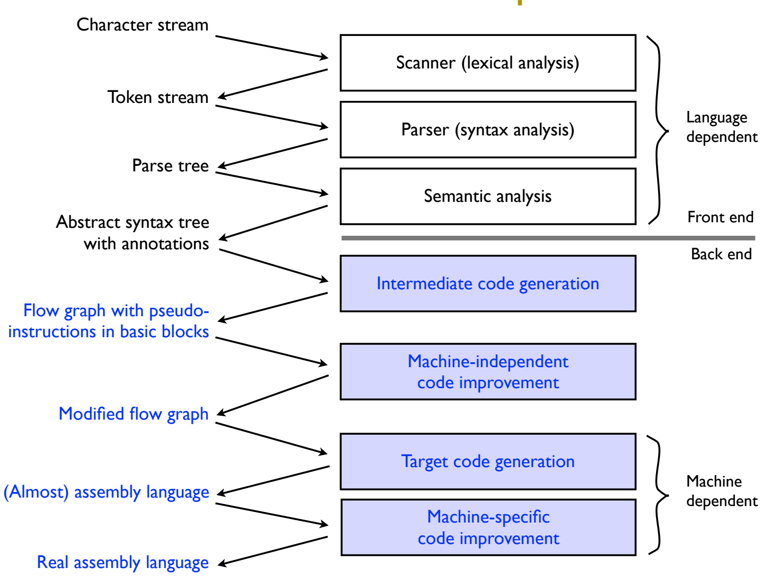
Phases of compilation

- Front end
  - analyses source code
  - is language dependent
  - fairly uniform
- Back end
  - produces target code
  - machine dependent
  - very non-uniform
- Intermediate
  - between the front and back end
  - independent of language and machine

Structure of compiler



Structure of compiler



Control flow graph

- Nodes
  - correspond to “basic blocks” of the syntax tree
  - a basic block is a maximal-length sequence of operations with no branches in or out
- Edges
  - represent interblock control flow
- Operations in blocks
  - instructions of an idealised machine, with an unlimited number of registers (“virtual” registers).

Machine-independent code improvement

- Within nodes
  - eliminate redundant loads, stores and arithmetic computations
- Between nodes
  - remove unnecessary repeated computations
    - e.g. an expression in a loop, the value of which will not change
  - these improvements can cause restructuring of the graph

Target code generation

- Translates nodes into instructions of target machine
- Strings these together with appropriate branches generated from graph edges
- Still relies on virtual registers
- Target code generators can be generated automatically from a formal description of a target machine

Machine-specific code improvement

- Register allocation
  - maps unlimited virtual registers to physical registers
  - if there are insufficient physical registers, generates extra loads and stores to maintain virtual registers in real memory
- Instruction scheduling
  - reorder instructions to keep pipeline as full as possible

Intermediate Forms

- High-level
  - based on trees (e.g. syntax trees), directed acyclic graphs, or stacks
- Medium-level
  - control flow graphs containing pseudo-assembly language instructions, quadruples consisting of
    - two operands
    - an operator
    - a destination
- Low-level
  - assembly language of target machine.

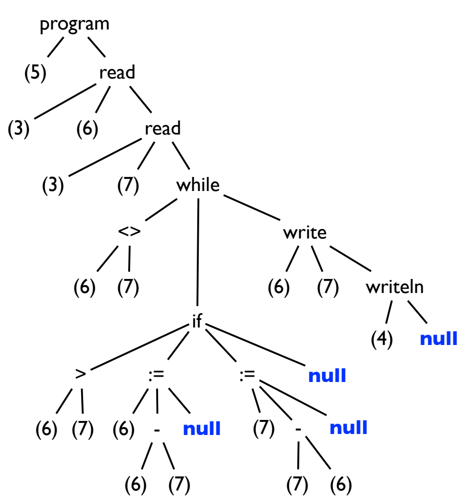
Choice of intermediate form

- Single language/single target
  - distinction between front and back end less clear
  - code improvement more at the lower levels
- Single language/several targets
  - do as much code improvement as possible on a high- or medium-level intermediate form
- Multilanguage compiler family
  - Intermediate form is independent of both language and target
  - For n languages, m machines - n front ends, m back ends instead of $n \times m$ different compilers
  - e.g. GCC compilers on Unix

Intermediate form representation

- Intermediate form needs linear representation for storing in a file
  - tree-based IFs
    - linearised by ordered traversal
  - control flow graphs
    - replace pointers with offsets from the beginning of the file

Syntax Tree and Symbol Table

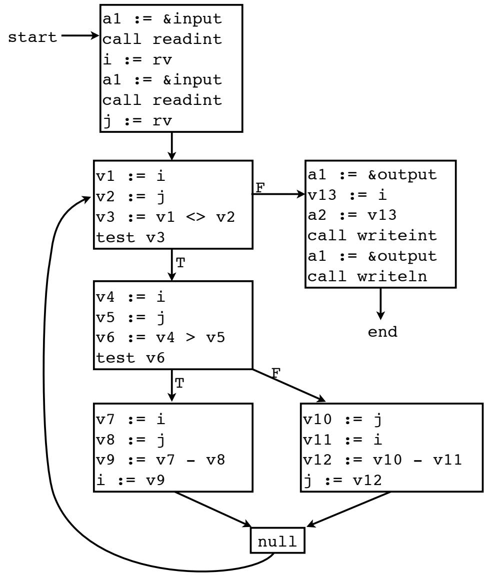



```
program gcd (input, output);
var i, j : integer;
begin
   read (i, j);
   while i <> j do
      if i > j then i := i - j
      else j := j - i;
   writeln (i)
end
```

Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

Control flow graph

```
program gcd (input, output);
var i, j : integer;
begin
   read (i, j);
   while i <> j do
      if i > j then i := i - j
      else j := j - i;
   writeln (i)
end
```

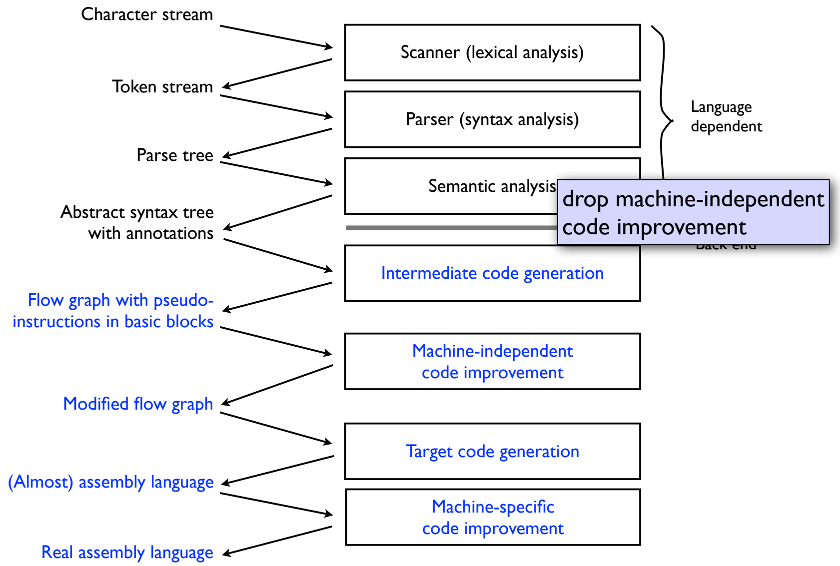


- v1,v2,... are general-purpose virtual registers
- a1, a2 are argument registers
- rv is a return-value register

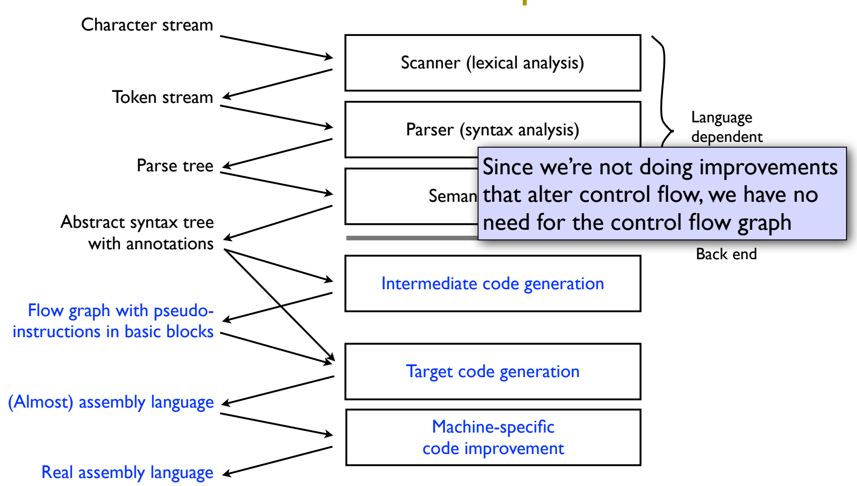
Code Generation

- Back end in our earlier compiler structure diagram is complex, so we focus on a simpler model, as follows…

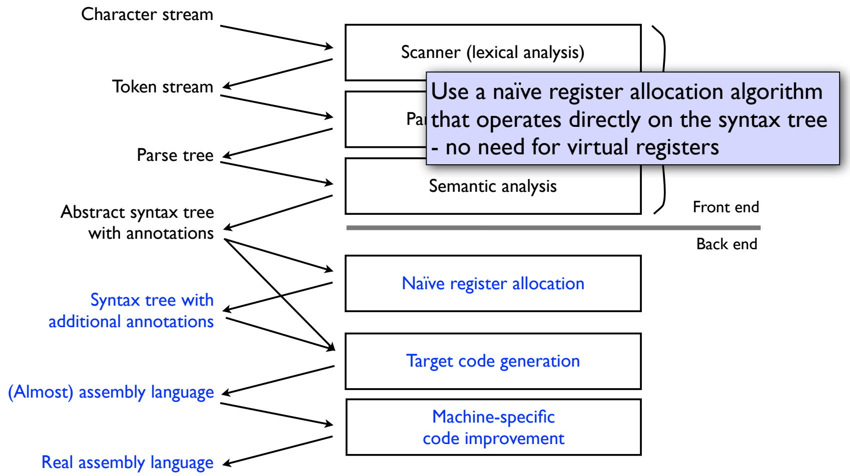
Structure of compiler



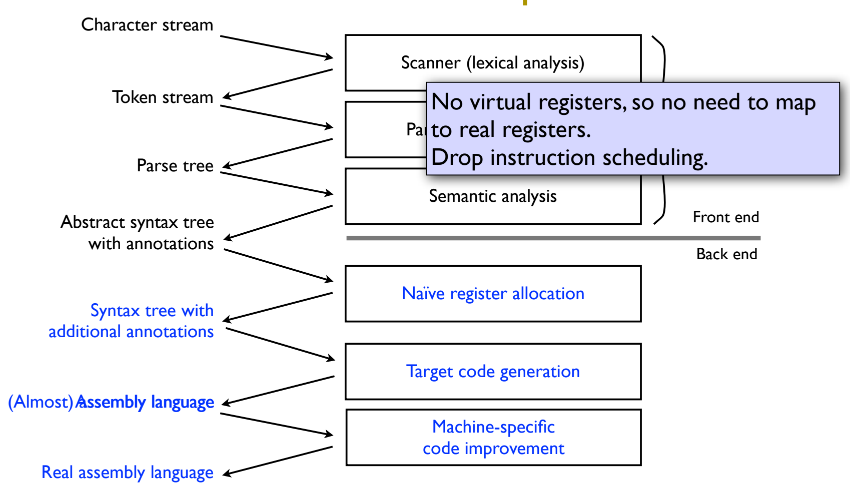
Structure of compiler



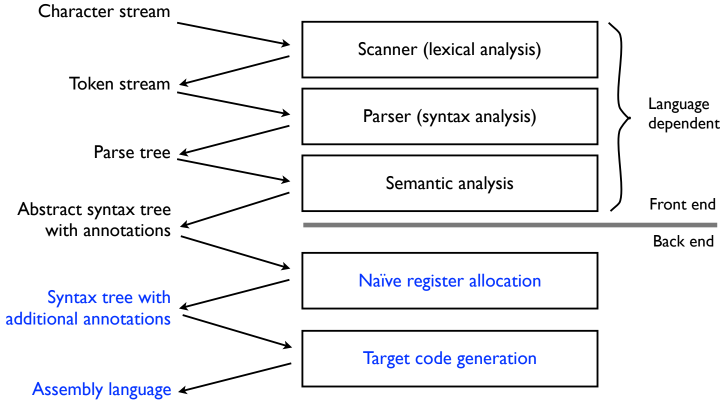
Structure of compiler



Structure of compiler



Structure of compiler



Code Generation

- From symbol table
  - Code for storage management
    - stack frame offsets for local variables and parameters
    - space for global variables

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

```
    .data
i:  .word
j:  .word
    .text
```

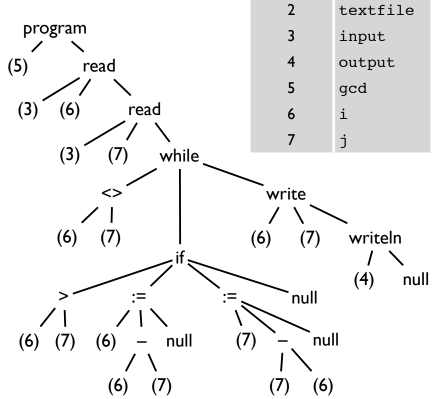
Syntax tree to code

- Uses a *tree grammar*
  - similar to a CFG, but describes the structure of syntax trees
  - used, with addition of attributes and rules, to annotate a syntax tree
    - see earlier discussion about uses for attribute grammars
- "A : B" on the LHS of a production means that A is a kind of B, and can appear anywhere a B is expected in a RHS

Syntax tree to code

- Tree grammar for the GCD example
  - Synthesized attributes
    - code - in program, expr, stmt
    - stp - in id, points to symbol table entry for identifier
    - reg - in expr, points to register
  - Inherited attribute
    - nfree_reg - in expr, stmt, is the next register available
  - Registers
    - assume a fixed and limited set
    - special purpose, for arguments (a1,a2…), return value (rv), stack pointer (sp) etc.
    - general purpose: r1,…,*rk*

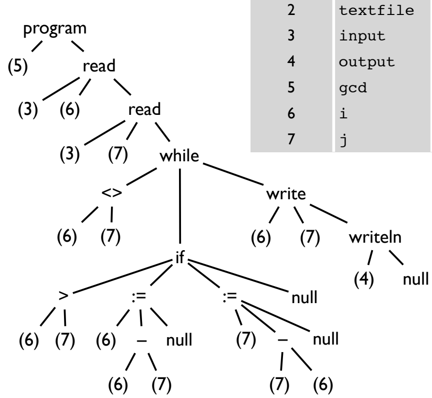
Syntax tree to code



Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

program → id stmt

- stmt.nfree_reg := 0
- program.code := [“main:”] + stmt.code + [“goto exit”]
- program.name := id.stp → name

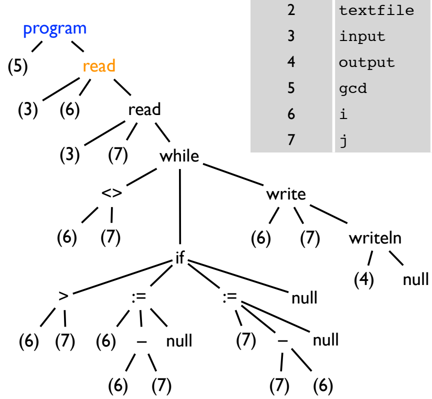


Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

program → id stmt

- stmt.nfree_reg := 0
- program.code := ["main:"] + stmt.code + ["goto exit"]
- program.name := id.stp → name

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

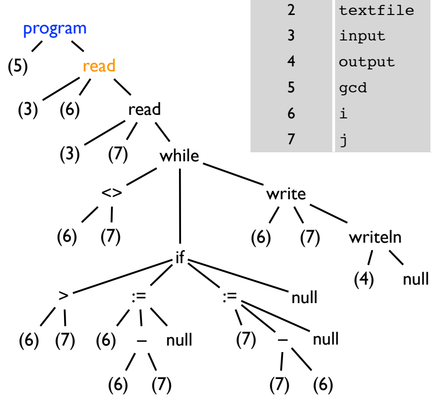


stmt		
	nfree_reg	code
	0	

program → id stmt

- stmt.nfree_reg := 0
- program.code := ["main:"] + stmt.code + ["goto exit"]
- program.name := id.stp → name

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1



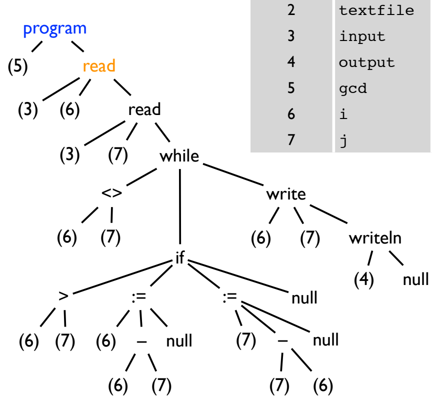
stmt		
	nfree_reg	code
	0	

program		
	name	code
		main: stmt.code goto exit

program → id stmt

- stmt.nfree_reg := 0
- program.code := [“main:”] + stmt.code + [“goto exit”]
- program.name := id.stp → name

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1



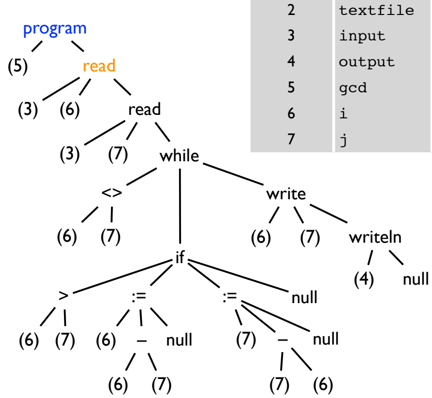
stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: stmt.code goto exit

read:stmt$_1$ → id$_1$ id$_2$ stmt$_2$

- stmt$_2$.nfree_reg := stmt$_1$.nfree_reg
- stmt$_1$.code := [“a1 := &” id$_1$.stp→name]
  + [“call” if id$_2$.stp→type = int then “readint” else …]
  + [id$_2$.stp→name “:= rv”] + stmt$_2$.code

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1



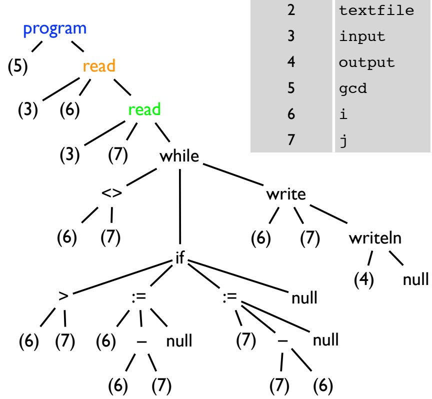
stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: stmt.code goto exit

read:stmt$_1$ → id$_1$ id$_2$ stmt$_2$

- stmt$_2$.nfree_reg := stmt$_1$.nfree_reg
- stmt$_1$.code := ["a1 := &" id$_1$.stp→name]
  + ["call" if id$_2$.stp→type = int then "readint" else …]
  + [id$_2$.stp→name ":= rv"] + stmt$_2$.code

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1



stmt

	nfree_reg	code
	0	

stmt

	nfree_reg	code
	0	

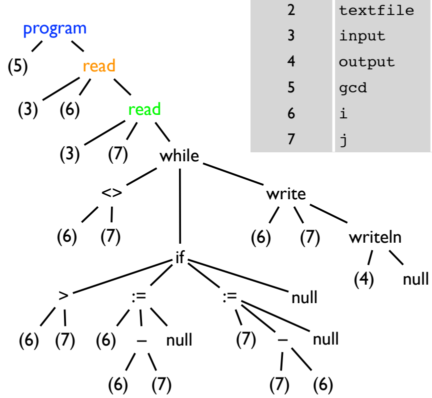
program

	name	code
	gcd	main: stmt.code goto exit

read:stmt$_1$ → id$_1$ id$_2$ stmt$_2$

- stmt$_2$.nfree_reg := stmt$_1$.nfree_reg
- stmt$_1$.code := ["a1 := &" id$_1$.stp→name]
  + ["call" if id$_2$.stp→type = int then "readint" else …]
  + [id$_2$.stp→name ":= rv"] + stmt$_2$.code

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1



stmt

	nfree_reg	code
	0	

stmt

	nfree_reg	code
	0	

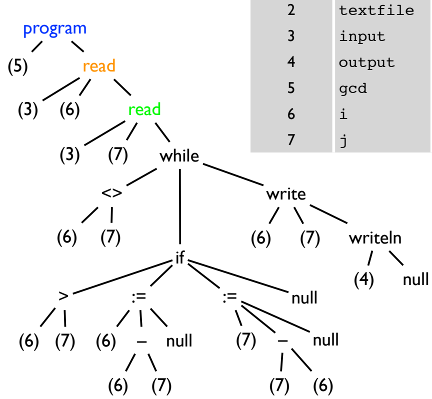
program

	name	code
	gcd	main: stmt.code goto exit

read:$stmt_1 \rightarrow id_1\ id_2\ stmt_2$

- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_1$.code := ["a1 := &" id_1.stp→name] + ["call" if id_2.stp→type = int then "readint" else …] + [id_2.stp→name ":= rv"] + $stmt_2$.code

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

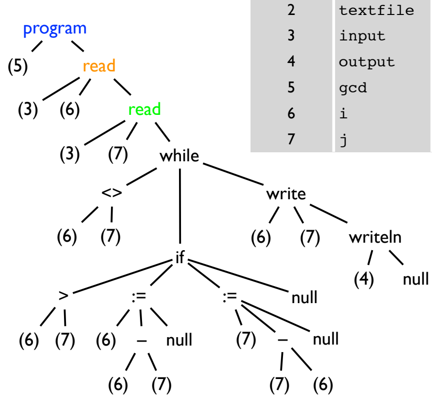


stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	a1 := &input call readint i := rv stmt.code

program		
	name	code
	gcd	main: stmt.code goto exit

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1



stmt		
	nfree_reg	code
	0	

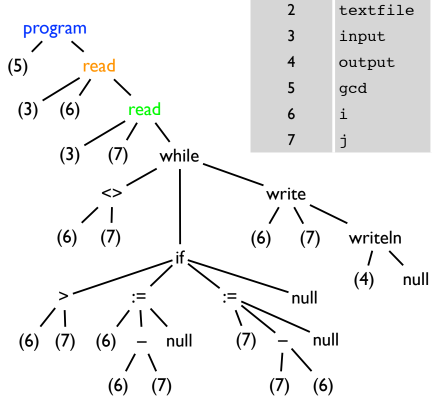
stmt		
	nfree_reg	code
	0	a1 := &input call readint i := rv stmt.code

program		
	name	code
	gcd	main: stmt.code goto exit

stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	a1 := &input call readint i := rv stmt.code

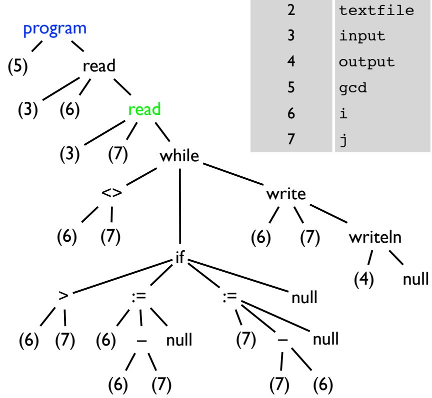
program		
	name	code
	gcd	main: stmt.code goto exit



Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv stmt.code goto exit



Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

$read{:}stmt_1 \rightarrow id_1\ id_2\ stmt_2$

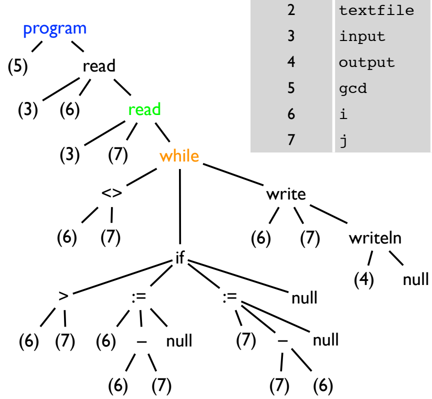
- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_1$.code := ["a1 := &" id_1.stp→name]
  + ["call" if id_2.stp→type = int then "readint" else ...]
  + [id_2.stp→name ":= rv"] + $stmt_2$.code

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv stmt.code goto exit



$read{:}stmt_1 \rightarrow id_1\ id_2\ stmt_2$

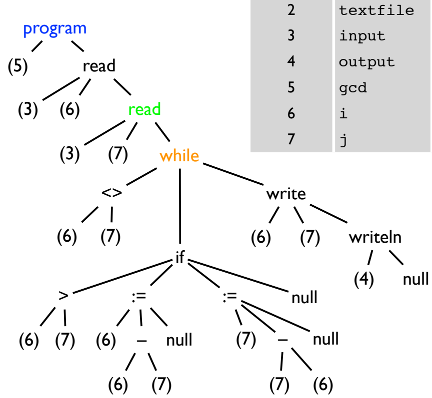
- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_1$.code := ["a1 := &" id_1.stp→name]
  + ["call" if id_2.stp→type = int then "readint" else ...]
  + [id_2.stp→name ":= rv"] + $stmt_2$.code

Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv stmt.code goto exit



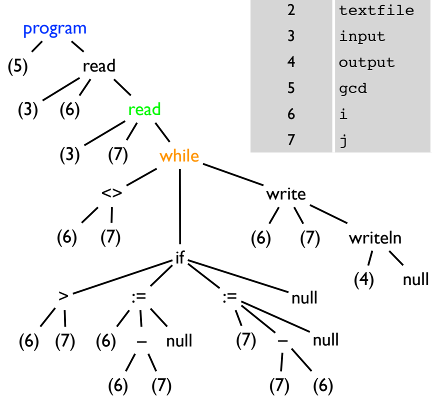
$read{:}stmt_1 \rightarrow id_1\ id_2\ stmt_2$

- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_1$.code := [“a1 := &” id_1.stp→name] + [“call” if id_2.stp→type = int then “readint” else …] + [id_2.stp→name “:= rv”] + $stmt_2$.code

stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	a1 := &input call readint j := rv stmt.code

program		
	name	code
	gcd	main: a1 := &input call readint i := rv stmt.code goto exit

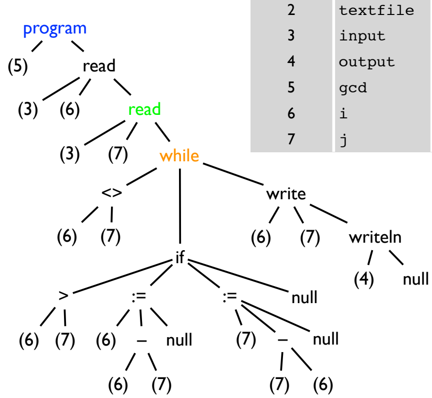


Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	a1 := &input call readint j := rv stmt.code

program		
	name	code
	gcd	main: a1 := &input call readint i := rv stmt.code goto exit

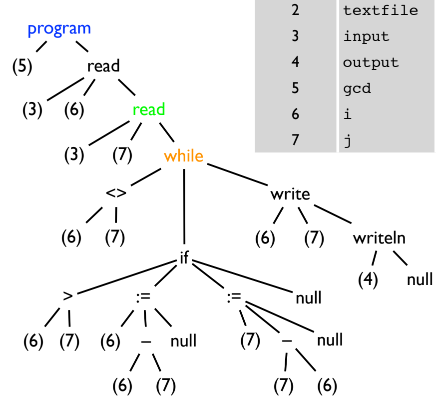


Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

stmt		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	a1 := &input call readint j := rv stmt.code

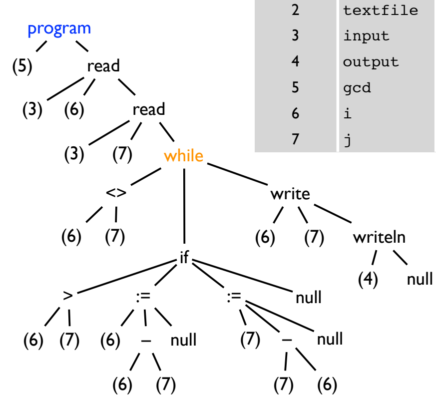
program		
	name	code
	gcd	main: a1 := &input call readint i := rv stmt.code goto exit



Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit



Index	Symbol	Type
1	`integer`	type
2	`textfile`	type
3	`input`	2
4	`output`	2
5	`gcd`	program
6	`i`	1
7	`j`	1

while:$stmt_1$ → expr $stmt_2$ $stmt_3$

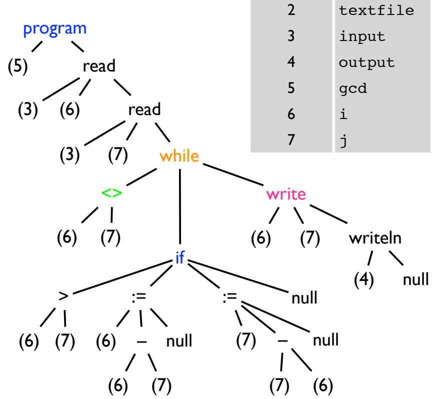
- expr.nfree_reg := $stmt_1$.nfree_reg
- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_3$.nfree_reg := $stmt_1$.nfree_reg
- L1 := new_label() ; L2 := new_label();
  $stmt_1$.code := [“goto” L1] + [L2 “:”] + $stmt_2$.code + [L1 “:”] + expr.code + [“if” expr.reg “goto” L2] + $stmt_3$.code

Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

expr		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit



while:$stmt_1$ → expr $stmt_2$ $stmt_3$

- expr.nfree_reg := $stmt_1$.nfree_reg
- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_3$.nfree_reg := $stmt_1$.nfree_reg
- L1 := new_label() ; L2 := new_label();
  $stmt_1$.code := [“goto” L1] + [L2 “:”] + $stmt_2$.code + [L1 “:”] + expr.code + [“if” expr.reg “goto” L2] + $stmt_3$.code

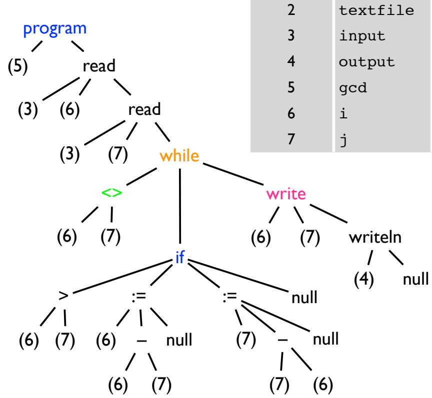
Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

stmt		
	nfree_reg	code
	0	

expr		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit



while:stmt$_1$ → expr stmt$_2$ stmt$_3$

- expr.nfree_reg := stmt$_1$.nfree_reg
- stmt$_2$.nfree_reg := stmt$_1$.nfree_reg
- stmt$_3$.nfree_reg := stmt$_1$.nfree_reg
- L1 := new_label() ; L2 := new_label();
  stmt$_1$.code := ["goto" L1] + [L2 ":"] + stmt$_2$.code + [L1 ":"] +
  expr.code + ["if" expr.reg "goto" L2] + stmt$_3$.code

stmt		
	nfree_reg	code
	0	

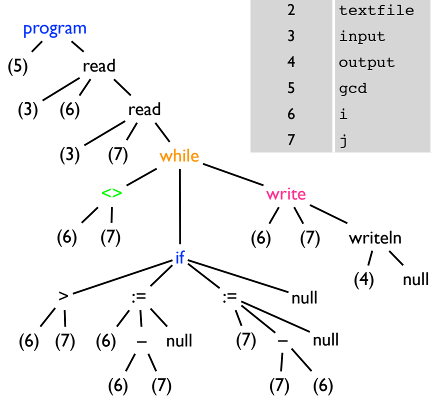
stmt		
	nfree_reg	code
	0	

expr		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit

Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1



while:stmt$_1$ → expr stmt$_2$ stmt$_3$

- expr.nfree_reg := stmt$_1$.nfree_reg
- stmt$_2$.nfree_reg := stmt$_1$.nfree_reg
- stmt$_3$.nfree_reg := stmt$_1$.nfree_reg
- L1 := new_label() ; L2 := new_label();
  stmt$_1$.code := ["goto" L1] + [L2 ":"] + stmt$_2$.code + [L1 ":"] +
  expr.code + ["if" expr.reg "goto" L2] + stmt$_3$.code

stmt		
	nfree_reg	code
	0	

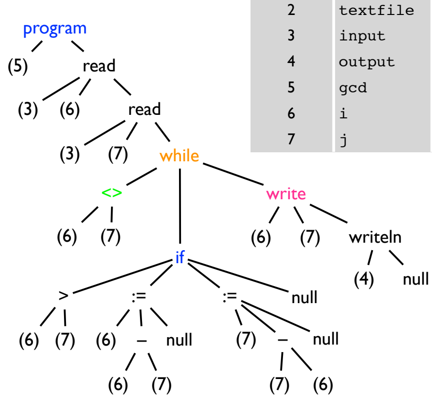
stmt		
	nfree_reg	code
	0	

expr		
	nfree_reg	code
	0	

stmt		
	nfree_reg	code
	0	

program		
	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit

Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1



	nfree_reg	code
	0	

stmt

	nfree_reg	code
	0	

expr

	nfree_reg	code
	0	

stmt

	nfree_reg	code
	0	goto x1 x2: stmt.code x1: expr.code if expr.reg goto x2 stmt.code

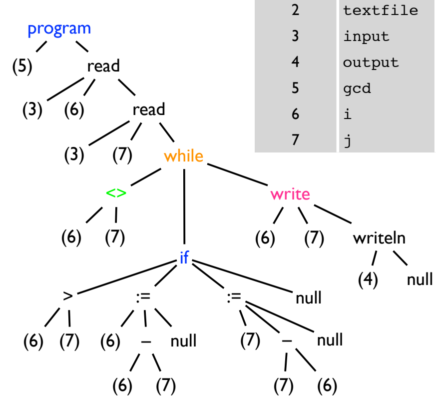
program

	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit

while:$stmt_1$ → expr $stmt_2$ $stmt_3$

- expr.nfree_reg := $stmt_1$.nfree_reg
- $stmt_2$.nfree_reg := $stmt_1$.nfree_reg
- $stmt_3$.nfree_reg := $stmt_1$.nfree_reg
- L1 := new_label() ; L2 := new_label();
  $stmt_1$.code := [“goto” L1] + [L2 “:”] + $stmt_2$.code + [L1 “:”] + expr.code + [“if” expr.reg “goto” L2] + $stmt_3$.code

Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1



	nfree_reg	code
	0	

stmt

	nfree_reg	code
	0	

expr

	nfree_reg	code
	0	

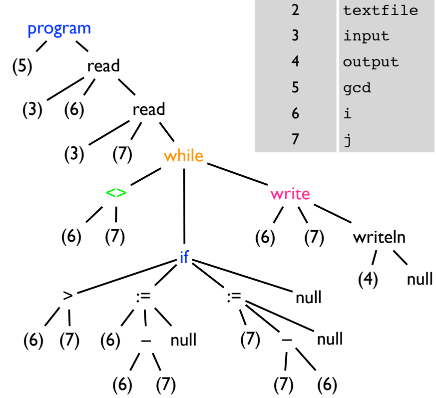
stmt

	nfree_reg	code
	0	goto x1 x2: stmt.code x1: expr.code if expr.reg goto x2 stmt.code

program

	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit

Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1



expr

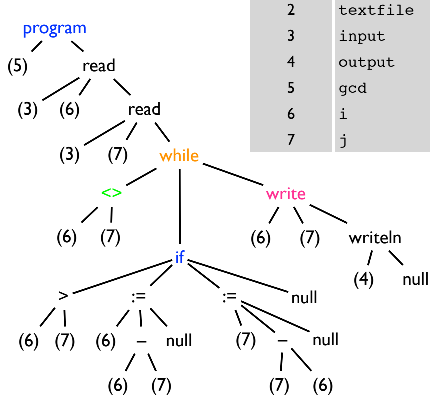
	nfree_reg	code
	0	

stmt

	nfree_reg	code
	0	goto x1 x2: stmt.code x1: expr.code if expr.reg goto x2 stmt.code

program

	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv stmt.code goto exit



Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

stmt

	nfree_reg	code
	0	

stmt

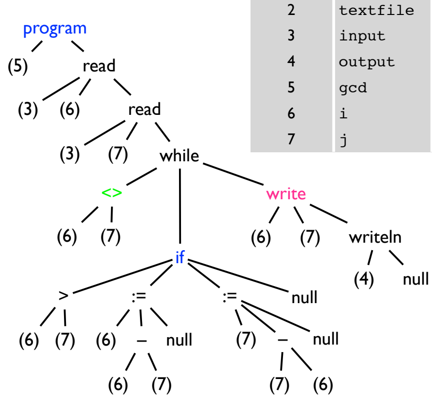
	nfree_reg	code
	0	

expr

	nfree_reg	code
	0	

program

	name	code
	gcd	main: a1 := &input call readint i := rv a1 := &input call readint j := rv goto x1 x2: stmt.code x1: expr.code if expr.reg goto x2 stmt.code goto exit



Index	Symbol	Type
1	integer	type
2	textfile	type
3	input	2
4	output	2
5	gcd	program
6	i	1
7	j	1

Register allocation

- Rule evaluation, for each subtree
  - determine registers to be used
  - generate code
- In naïve register allocation, use a register stack
- If you run out of physical registers, spill registers into memory

Address space organization

- Two kinds of object code
  - relocatable
    - input to linker
    - several files linked together into an executable program
  - executable
    - input to loader
    - can be loaded into memory and run

A relocatable object file

- Includes the following
  - import table
    - identifies instructions that refer to named locations presumed to lie in other files.
  - relocation table
    - identifies instructions that refer to locations in current file, which must be modifed at link time to reflect the position of current file in the executable program
  - export table
    - lists names and addresses of locations in current file that can be referred to in other files.

Contents of a running program

- code
- constants
- initialised data
- uninitialised data
- stack
  - small initial size, grown by the OS
- heap
  - small initial size, grown on demand
- files - mapped into memory

Assembly

- Translates assembly code into executable code
  - replaces opcodes and operands with machine language encodings
  - replaces symbolic names with actual addresses
- Modern assemblers may perform some machine-specific code improvement
  - instruction scheduling
  - register allocation
  - peephole optimisation
    - fixes suboptimal patterns of instructions within a small window

Constructing instructions

- Many assemblers
  - extend the instruction set in minor ways to make assembly language easier for humans to read
    - e.g. pseudo-instructions in MIPS
  - have directives for
    - segment switching, e.g. .text in MIPS
    - data generation, e.g. .byte, .half, .word, .float
    - symbol identification, e.g. .globl
    - alignment, e.g. .align

Assigning addresses to names

- Pass 1
  - Identify internal and external (imported) symbols
    - symbol in `.globl` directive is put in export table
    - any symbol mentioned in an instruction but not defined is put in import table, with instruction offset
    - any instruction or data with a value depending on placement of file into executable program is put in the relocation table
  - assign locations to the internal symbols
- Pass 2
  - Produce object code

Linker

- Joins together compilation units
  - static linker - prior to program execution
  - dynamic linker - during execution
- Two subtasks
  - Relocation
  - Resolution of external references

Relocation

- Phase 1
  - gather compilation units together
  - choose an order for them in memory
  - note addresses of each unit in memory
- Phase 2
  - replace unresolved external references with addresses
  - modify instructions that need to be relocated to reflect addresses of their units

The End