Comp 104: Operating Systems Concepts

Introduction to Compilers

Today

- Compilers
  - Definition
  - Structure
  - Passes
  - Lexical Analysis
    - Symbol table
      - Access methods

Compilers

- Definition:
  - A compiler is a program which translates a high-level source program into a lower-level object program (target)

History

- Late 1940ies (post-von Neumann)
  - Programs were written in machine code
    - `C7 06 0000 0002` (move the number "2" to location 0000 (hex)
    - Highly complex, tedious and prone to error
- Assemblers appeared
  - Machine instructions given as mnemonics
    - `MOV X,2` (assuming X has the value 0000 (hex))
    - Greatly improved the speed and accuracy of writing code
    - But still non-trivial, and non-portable to new processors
- Needed a mathematical notation
  - Fortran appeared between 1954-57
    - `X = 2`
    - Exploited context free grammars (Chomsky) and finite state automatata…

Compiler

- Responsible for converting source code into executable code.
    - Analyses the code to determine the functionality
    - Synthesises executable code for a given processor
    - Optimises code to improve performance, or exploit specific processor instructions
- Assumes various data structures:
  - Tokens
    - Variables, language keywords, syntactic constructs etc
  - Symbol Table
    - Relates user defined entities (variables, methods, classes etc) with their associated values or internal structures
  - Literal Table
    - Stores constants, strings, etc. Used to reduce the size of the resulting code
  - Syntax/Parse Tree
    - The resulting structure formed through the analysis of the code
  - Intermediate Code
    - Intermediate representation between different phases of the compilation

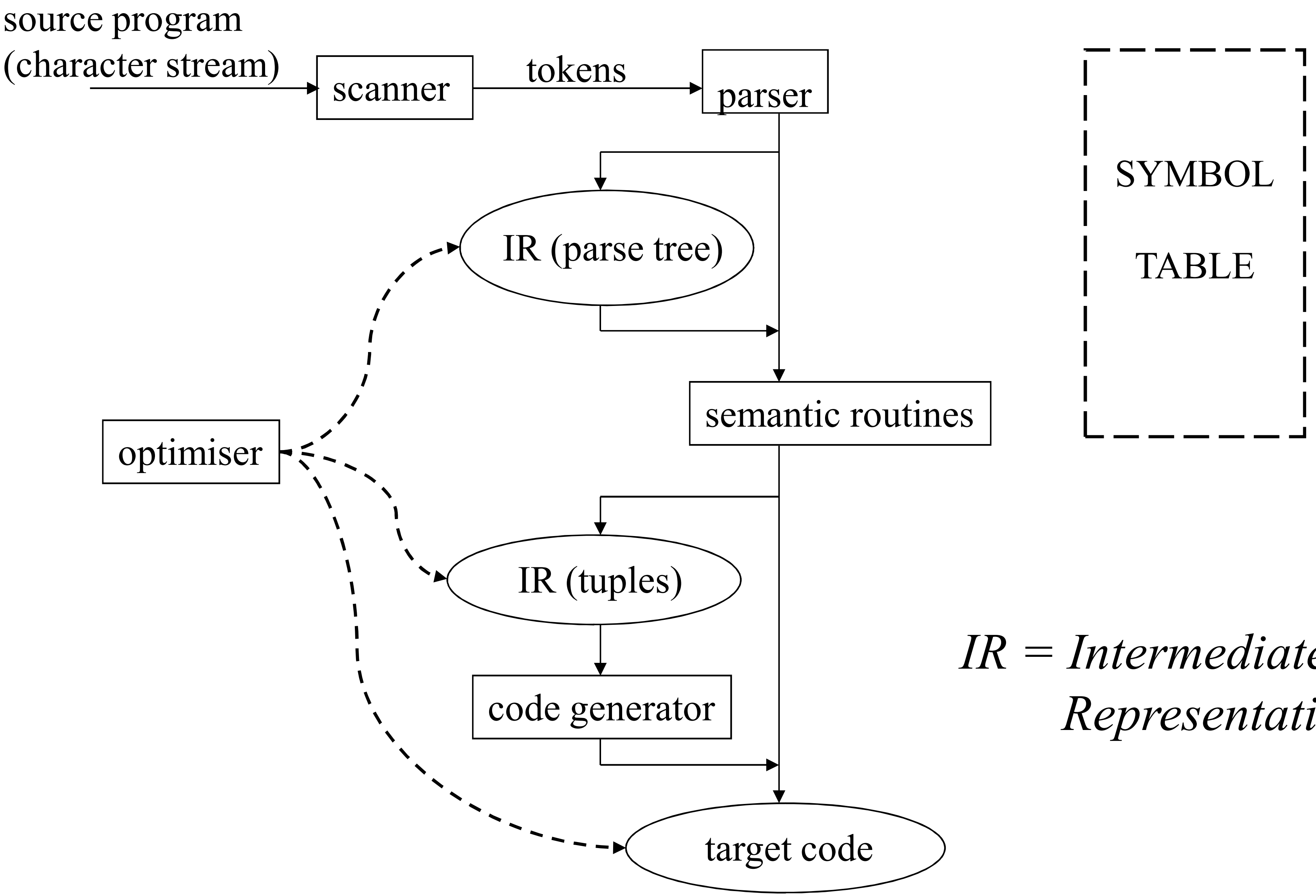
Phases and other tools

- **Interpreters:**
  - Unlike compilers, code is executed immediately
    - Slow execution, used more for scripting or functional languages
- **Assemblers:**
  - Constructs final machine code from processor specific Assembly code
    - Often used as last phase of a compilation process to produce binary executable.
- **Linkers:**
  - Collates separately compiled objects into a single file, including shared library objects or system calls.
- **Preprocessors:**
  - Called prior to the compilation process to perform macro substitutions
    - E.g. RATFOR preprocessor, or cpp for C code…
- **Profilers:**
  - Collects statistics about the behaviour of a program and can be used to improve the performance of the code.

Analysis and Synthesis

- Analysis:
  - checks that program constructs are legal and meaningful
  - builds up information about objects declared
- Synthesis:
  - takes analysed program and generates code necessary for its execution
- Compilation based on language definition, which comprises:
  - syntax
  - semantics

Compiler Structure



IR = Intermediate Representation

Compiler Organisation

- Each of compiler tasks described previously (in Compiler Structure) is a phase
- Phases can be organised into a number of passes
  - a pass consists of one or more phases acting on some representation of the complete program
  - representations produced between source and target are Intermediate Representations (IRs)

Single Pass Compilers

- One pass compilers very common because of their simplicity
- No IRs: all phases of compiler interleaved
- Compilation driven by parser
- Scanner acts as subroutine of parser, returning a token on each call
- As each phrase recognised by parser, it calls semantic routines to process declarations, check for semantic errors and generate code
- Code not as efficient as multi-pass

Multi-Pass Compilers

- Number of passes depends on number of IRs and on any optimisations
- Multi-pass allows complete separation of phases
  - more modular
  - easier to develop
  - more portable
- Main forms of IR:
  - Abstract Syntax Tree (AST)
  - Intermediate Code (IC)
    - Postfix
    - Tuples
    - Virtual Machine Code

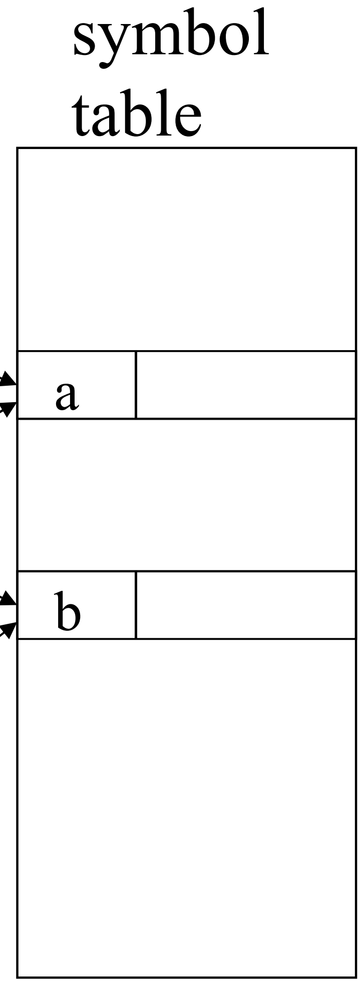
The Scanner (Lexical Analyser)

- Converts groups of characters into tokens (lexemes)
  - tokens usually represented as integers
  - white space and comments are skipped
- Each token may be accompanied by a value
  - could be a pointer to further information
- As identifiers encountered, entered into a symbol table
  - used to collect info. about declared objects
- Scanners often hand-coded for efficiency, but may be automatically generated (e.g. Lex)

Example

```
begin
  int a; float b;
  a = 1; b = 1.2;
  a = b + 1;
  print (a * 2);
end
```

	TOKEN	VALUE
`begin`	beginsymb	
`int a;`	intsymb	
	iden	@
	semisymb	
`float b;`	floatsymb	
	iden	@
	semisymb	
`a = 1;`	iden	@
	assignsymb	
	integer	1
	semisymb	
`b = 1.2;`	iden	@
	assignsymb	
	float	1.2



Symbol Table Access

- The symbol table is used by most compiler phases
  - Even used post-compilation (debugging)
- Structure of table and algorithms used can make difference between a slow and fast compiler
- Methods:
  - Sequential lookup
  - Binary chop and binary tree
  - Hash addressing
  - Hash chaining

Sequential Lookup

- Table is just a vector of names
- Search sequentially from beginning
- If name not found, add to end
- Advantages:
  - Very simple to implement
- Disadvantages:
  - Inefficient
  - For table with N names, requires N/2 comparisons on average
  - Can slow down a compiler by a factor of 10 or more

Binary Chop

- Keep names in alphabetical order
- To find name:
  - Compare with middle element to determine which half
  - Compare with middle element again to narrow down to quarter, etc.
- Advantage:
  - Much more efficient than sequential
  - $\log_2 N$-1 comparisons on average
- Disadvantage:
  - Adding a new name means shifting up every name above it

Question

- If the symbol table for a compiler is size 4096, how many comparisons on average need to be made when performing a lookup using the binary chop method?

  a) 2
  b) 11
  c) 12
  d) 16
  e) 31

Answer: b
11 – as there are $\log_2 N-1$ comparisons on average

Binary Tree

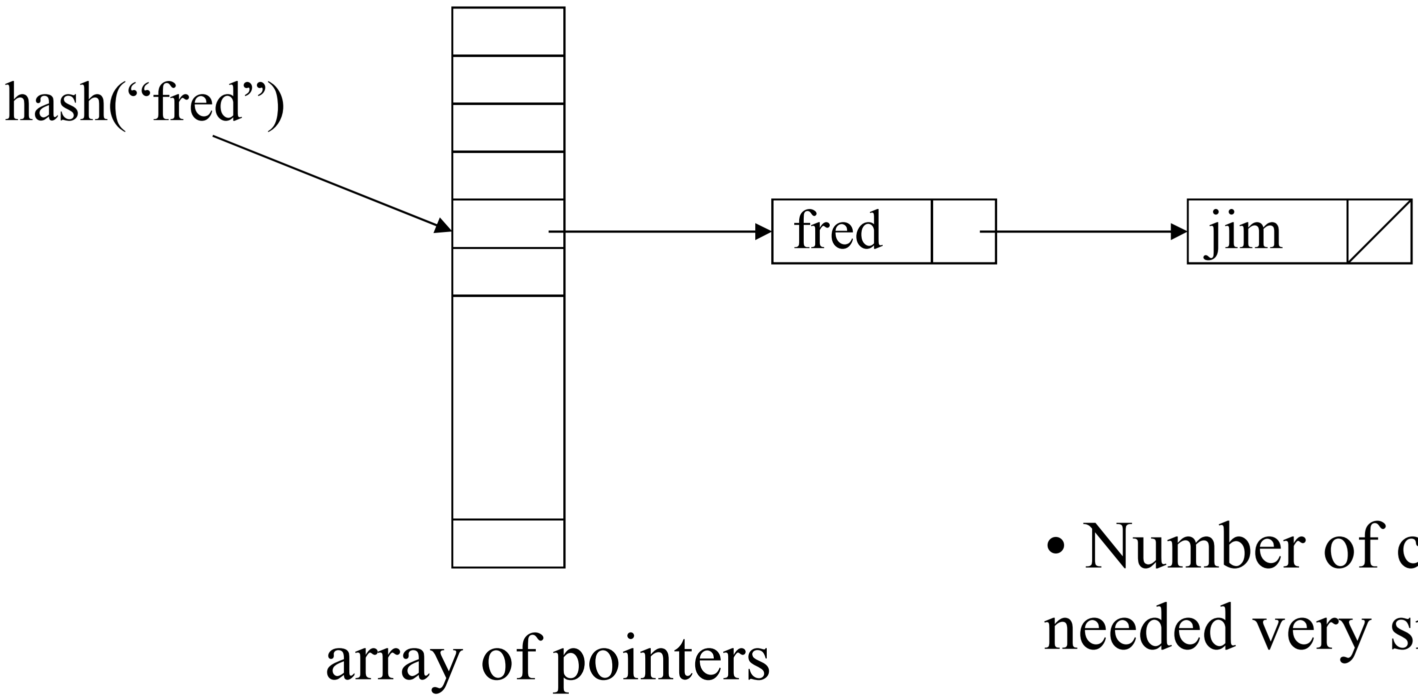
- Each node contains pointer to 2 sub-trees
  - Left sub-tree contains all names < current
  - Right sub-tree has all names >= current
- Advantages:
  - In best case, search time can be as good as binary chop
  - Adding a new name is simple and efficient
- Disadvantages:
  - Efficiency depends on how balanced the tree is
  - Tree can easily become unbalanced
  - In worst case, method as bad as sequential lookup!
  - May need to do costly re-balancing occasionally

Hash Addressing

- To determine position in table, apply a hash function, returning a hash key
  - Example fn: Sum of character codes modulo N, where N is table size (prime)
- Advantages:
  - Can be highly efficient
  - Even similar names can generate totally different hash keys
- Disadvantages:
  - Requires hash function producing good distribution
  - Possibility of collisions
  - May require re-hashing mechanism, possibly multiple times

Hash Chaining

- As before, but link together names having same hash key



- Number of comparisons needed very small

Question

- Concerning compilation, which of the following is NOT a method for symbol table access?

  a) Sequential lookup
  b) Direct lookup
  c) Binary chop
  d) Hash addressing
  e) Hash chaining

Answer: b
Direct Lookup

Reserved Words

- Words like ‘for’, ‘while’, ‘if’, etc. are reserved words
- Could use binary chop on a table of reserved words first; if not there, search symbol table
- Simpler to pre-hash all reserved words into the symbol table and use one lookup mechanism

Today

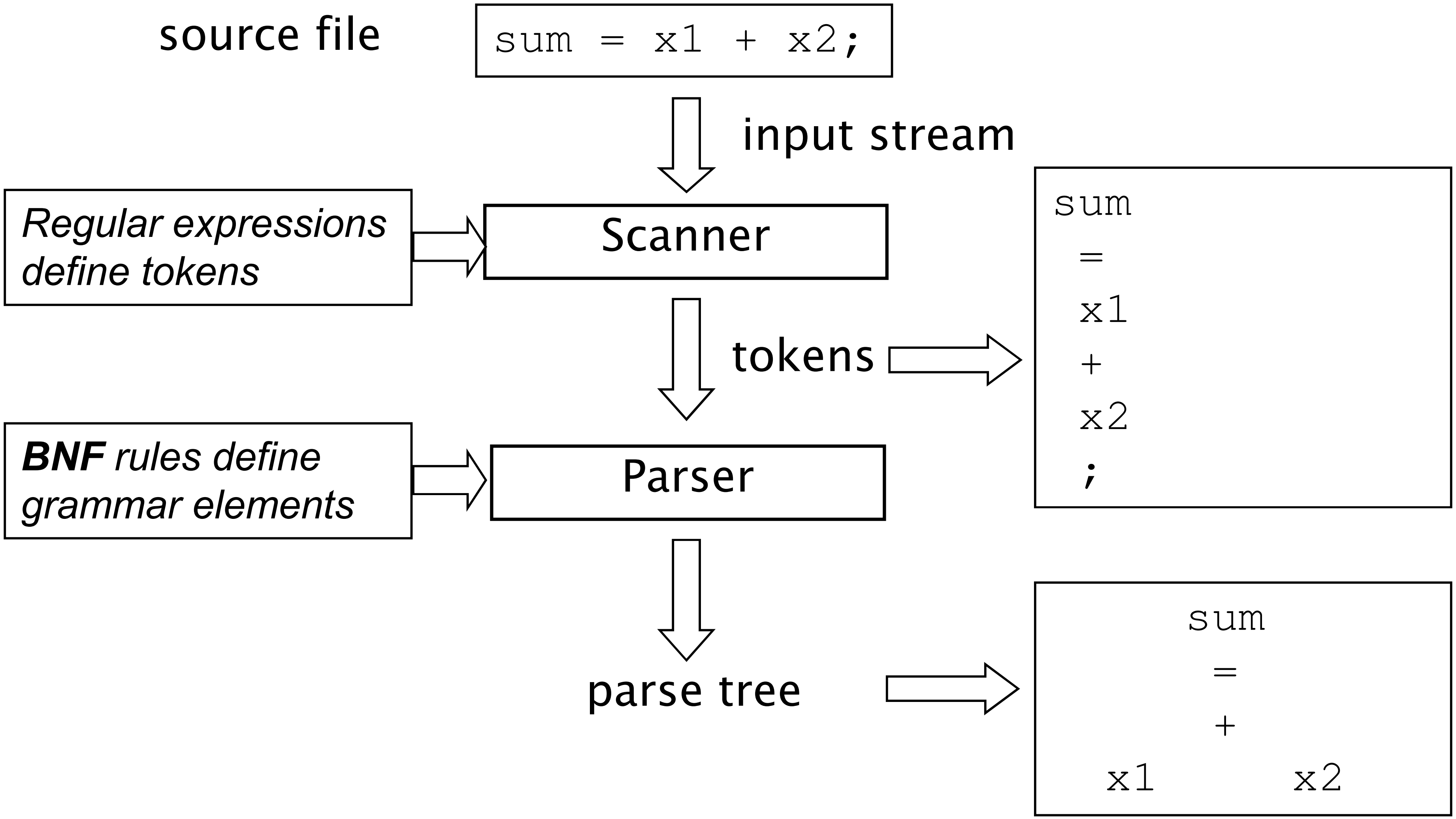
- Parsing
  - Parse Tree
  - Abstract syntax tree

Parser (Syntax Analyser)

- Reads tokens and groups them into units as specified by language grammar i.e. it recognises syntactic phrases

- Parser must produce good errors and be able to recover from errors

Scanning and Parsing



Syntax

- Defines the structure of legal statements in the language
- Usually specified formally using a context-free grammar (CFG)
- Notation most widely used is Backus-Naur Form (BNF), or extended BNF
- A CFG is written as a set of rules (productions)

Backus Naur Form

- **Backus Naur Form** (BNF) is a standard notation for expressing ***syntax*** as a set of grammar rules.
  - BNF was developed by Noam Chomsky, John Backus, and Peter Naur.
  - First used to describe Algol.
- BNF can describe any ***context-free grammar***.
  - Fortunately, computer languages are mostly context-free.

A Context-Free Grammar

A grammar is *context-free* if all the syntax rules apply regardless of the symbols before or after (the context).

Example:

(1) *sentence* => *noun-phrase* *verb-phrase* .

(2) *noun-phrase* => *article* *noun*

(3) *article* => `a` | `the`

(4) *noun* => `boy` | `girl` | `cat` | `dog`

(5) *verb-phrase* => *verb* *noun-phrase*

(6) *verb* => `sees` | `pets` | `bites`

Terminal symbols:

'`a`' '`the`' '`boy`' '`girl`' '`sees`' '`pets`' '`bites`'

A Context-Free Grammar

A sentence that matches the *productions* (1) - (6) is valid.

a girl sees a boy
a girl sees a girl
a girl sees the dog
the dog pets the girl
a boy bites the dog
a dog pets the boy
...

To eliminate unwanted sentences without imposing *context sensitive* grammar, specify semantic rules:

"a boy may not bite a dog"

Backus Naur Form

- *Grammar Rules or Productions:* define symbols.

assignment_stmt **::=** *id* = *expression* ;

The *nonterminal symbol being defined.*

The *definition* (production)

Nonterminal Symbols: anything that is defined on the left-side of some production.

***Terminal Symbols**:* things that are not defined by productions. They can be literals, symbols, and other *lexemes* of the language defined by lexical rules.

Identifiers: *id* ::= [A-Za-z_]\w*

Delimiters: ;

Operators: = + - * / %

Backus Naur Form (2)

- Different notations (same meaning):

  assignment_stmt ::= id = expression + term

  <assignment-stmt> => <id> = <expr> + <term>

  AssignmentStmt $\rightarrow$ *id = expression + term*

  ::=, =>, $\rightarrow$ mean "*consists* of" or "*defined as*"

- Alternatives (" | "):

```
expression  => expression + term
                 | expression - term
                 | term
```

- Concatenation:

```
number   => DIGIT number | DIGIT
```

Alternative Example

- The following BNF syntax is an example of how an arithmetic expression might be constructed in a simple language…

- Note the recursive nature of the rules

Syntax for Arithmetic Expr.

```
<expression> ::= <term> | <addop> <term> |<expression> <addop> <term>

<term> ::= <primary> | <term> <multop> <primary>

<primary> ::= <digit> | <letter> | ( <expression> )

<digit> ::= 0 | 1 | 2 |...| 9

<letter> ::= a | b | c |...| y | z

<multop> ::= * | /

<addop> ::= + | -
```

- Are the following expressions legal, according to this syntax?
  - i) -a
  - ii) b+c^(3/d)
  - iii) a*(c-(4+b))
  - iv) 5(9-e)/d

BNF rules can be recursive

```
expr  => expr + term
           | expr - term
           | term
term  => term * factor
           | term / factor
           | factor
factor => ( expr ) | ID | NUMBER
```

where the tokens are:

```
NUMBER := [0-9]+
ID     := [A-Za-z_][A-Za-z_0-9]*
```

Uses of Recursion

- **Repetition**

expr => *expr* + *term*

=> *expr* + *term* + *term*

=> *expr* + *term* + *term* + *term*

=> ***term* + ... + *term* + *term***

- Parser can recursively expand *expr* each time one is found
  - Could lead to arbitrary depth analysis
  - Greatly simplifies implementation

Example: The Micro Language

- To illustrate BNF parsing, consider an example imaginary language: the "*Micro*" language

1) A program is of the form

```
begin
        sequence of statements
end
```

2) Only statements allowed are
   - assignment
   - read (list of variables)
   - write (list of expressions)

Micro

3) Variables are declared implicitly
   – their type is integer

4) Each statement ends in a semi-colon

5) Only operators are +, -
   – parentheses may be used

Micro CFG

1.	<program>	::=	begin <stat-list> end
2.	<stat-list>	::=	<statement> { <statement> }
3.	<statement>	::=	id := <expr> ;
4.	<statement>	::=	read (<id-list>) ;
5.	<statement>	::=	write (<expr-list>) ;
6.	<id-list>	::=	id { , id }
7.	<expr-list>	::=	<expr> { , <expr> }
8.	<expr>	::=	<primary> { <addop> <primary> }
9.	<primary>	::=	(<expr>)
10.	<primary>	::=	id
11.	<primary>	::=	intliteral
12.	<addop>	::=	+
13.	<addop>	::=	-

1) A program is of the form

```
begin
  statements
end
```

2) Permissible statements:

- assignment
- read (list of variables)
- write (list of expressions)

3) Variables are declared implicitly their type is integer

4)Statements end in a semi-colon

5) Valid operators are +, - but can use parentheses

BNF

- Items such as <`program`> are non-terminals
  - require further expansion
- Items such as `begin` are terminals
  - correspond to language tokens
- Usual to combine productions using | (or)
  - e.g. <primary> ::= (<expr>) | id | intliteral

Parsing

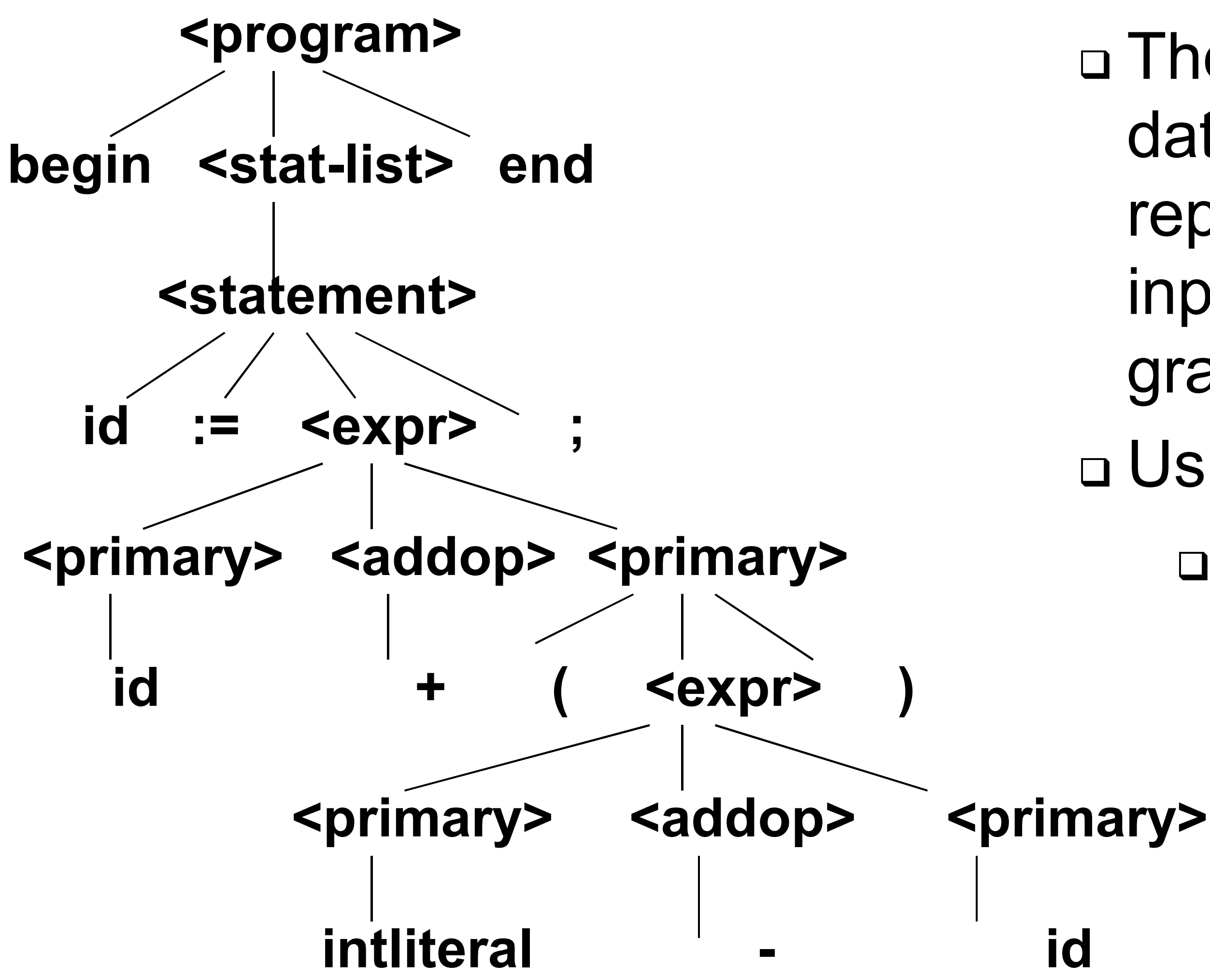
- Bottom-up
  - Look for patterns in the input which correspond to phrases in the grammar
  - Replace patterns of items by phrases, then combine these into higher-level phrases, and so on
  - Stop when input converted to single <program>
- Top-down
  - Assume input is a <program>
  - Search for each of the sub-phrases forming a <program>, then for each of the sub-sub-phrases, and so on
  - Stop when we reach terminals
- A program is syntactically correct iff it can be derived from the CFG

Example

Parse: **`begin  A := B + (10 – C); end`**

```
<program>
begin <stat-list> end                          (apply rule 1)
begin <statement> end                                     (2)
begin id := <expr> ; end                                  (3)
begin id := <primary> <addop> <primary>; end    (8)
begin id := <primary> + <primary> ; end              (12)
...
```

Parse Tree



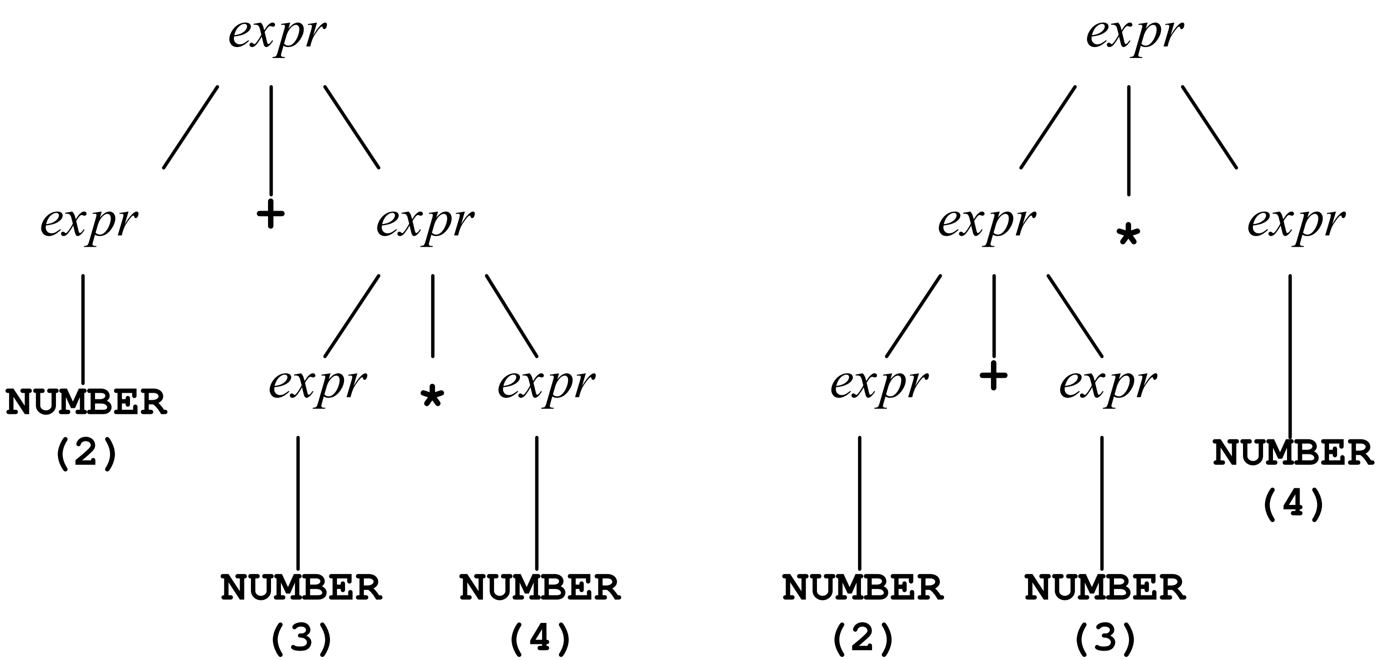
- The parser creates a data structure representing how the input is matched to grammar rules.
- Usually as a *tree.*
  - *Also called syntax tree or derivation tree*

Example of Ambiguity

- Grammar Rules:

  expr => ***expr*** + ***expr*** **|** ***expr*** ∗ ***expr***
  | (***expr***) **|** NUMBER

- Expression: 2 + 3 * 4
- Two possible parse trees:



Ambiguity

- Ambiguity can lead to inconsistent implementations of a language.
  - Ambiguity can cause infinite loops in some parsers.
  - Specification of a grammar should be <u>un</u>ambiguous!
- How to resolve ambiguity:
  - rewrite grammar rules to remove ambiguity
  - add some additional requirement for parser, such as "always use the left-most match first"

Semantics

- Specify meaning of language constructs
  - usually defined informally
- A statement may be syntactically legal but semantically meaningless
  - “colourless green ideas sleep furiously”
- Semantic errors may be
  - static (detected at compile time)
    e.g. a := ‘x’ + true;
  - dynamic (detected at run time)
    e.g. array subscript out of bounds

Semantics

- Also needed to generate appropriate code e.g. a = b
  - in Java and C, this means assign b to a
  - in Pascal and Ada, this means compare equality of a and b
  - hence, generate different code in each case

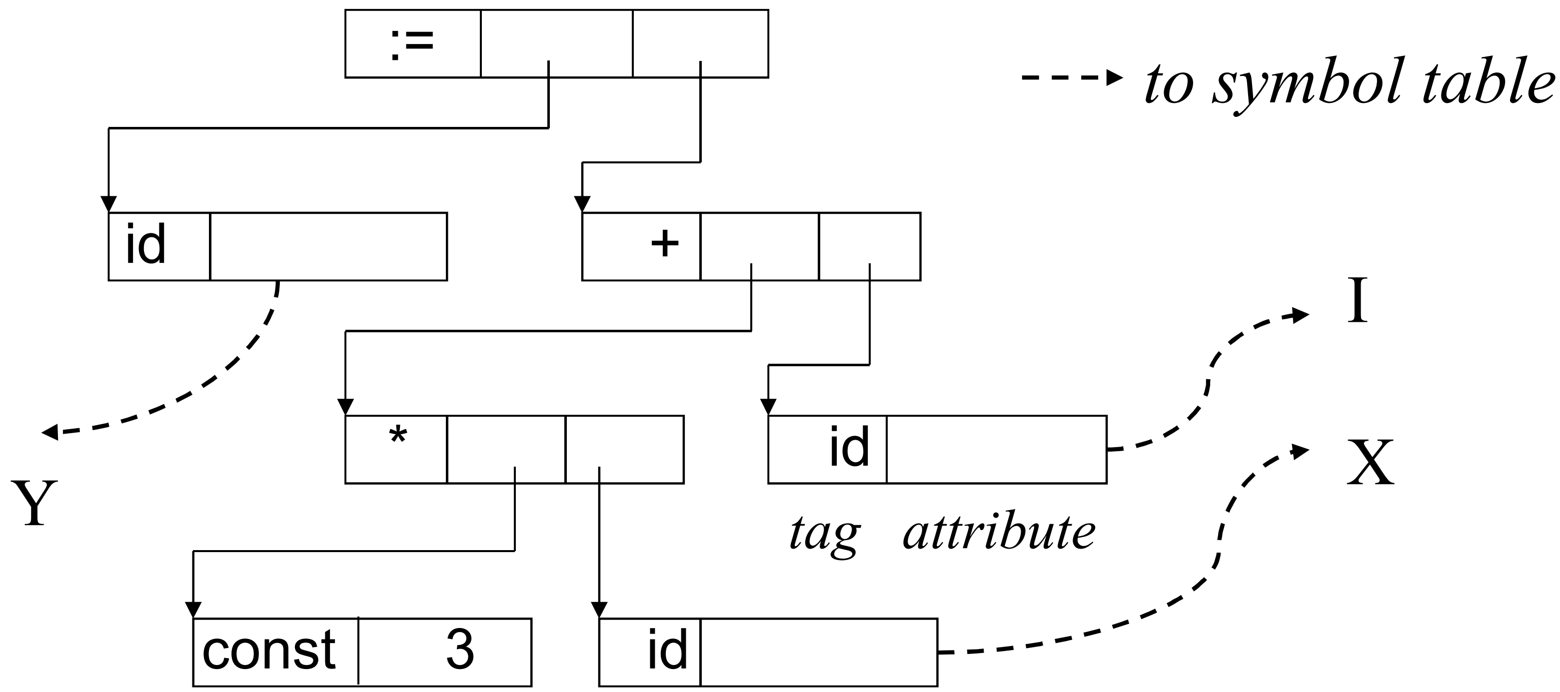
Semantic Routines

1) Semantic analysis
   - Completes analysis phase of compilation
   - Object descriptors are associated with identifiers in symbol table
   - Static semantic error checking performed

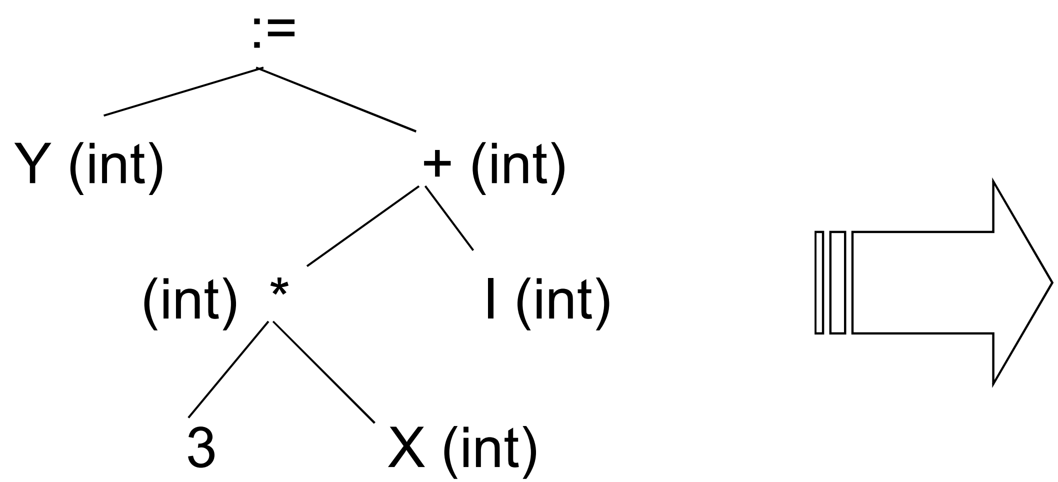
2) Semantic synthesis
   - Code generation

Abstract Syntax Tree (AST)

- More compact form of derivation tree
  - contains just enough info. to drive later phases
    e.g. Y := 3*X + I



Tree Walking




```
LOAD R1, #3
LOAD R2, X
MULT R1, R2
LOAD R2, I
ADD R1, R2
STORE R1, Y
```

- Advantage of AST is that order of traversal can be chosen
  - code generated in one-pass compiler corresponds to strictly fixed traversal of tree
    (hence, code not as good)

Code Optimisation

- Aim is to improve quality of target code
- Disadvantages
  - compiler more difficult to write
  - compilation time may double or triple
  - target code often bears little resemblance to unoptimised code
    - greater chance of translation errors
    - more difficult to debug programs

Optimisation Techniques

- Constant folding
  - can evaluate expressions involving constants at compile-time
  - aim is for the compiler to pre-compute (or remove) as many operations as possible

```
a := 3*16 - 2;
   ⇩
LOAD 1, #46
STORE 1, a
```

Techniques

- Global register allocation
  - analyse program to determine which variables are likely to be used most and allocate these to registers
  - good use of registers is a very important feature of efficient code
    - aided by architectures that provide an increased number of registers

Techniques

- Code deletion
  - identify and delete unreachable or dead code

```
boolean debug = false;
...
if (debug) {
    ...
}
```

No need to generate code for this

Techniques

- Common sub-expression elimination
  - avoid generating code for unnecessary operations by identifying expressions that are repeated

```
a := (b*c/5 + x) – (b*c/5 + y)
```

  - generate code for **`b*c/5`** only once

Techniques

- Code motion out of loops

```
for (int i=0; i <= n; i++) {
    x = a + 5;   //loop-invariant code
    Screen.println(x*i);
}
```

```
x = a + 5;
for (int i=0; i <= n; i++) {
    Screen.println(x*i);
}
```

Question

- What optimisation technique could be applied in the following examples?

  a = b^2

  a = a / 2

a) Constant Folding
b) Code Deletion
c) Common Sub-Expression Elimination
d) Strength Reduction
e) Global Register Allocation

Answer: d
Both expressions can be reduced by changing the operator:
a = b ^ 2 can be reduced to a = b * b
a = a / 2 is a right shift operation: a = a >> 1

Classification of Optimisations

- Optimisations can be classified according to their different characteristics
- Two useful classifications:
  - the period of the compilation process during which an optimisation can be applied
  - the area of the program to which the optimisation applies

Time of Application

- Optimisations can be performed at virtually every stage of the compilation process
  - e.g. constant folding can be performed during parsing
  - other optimisations might be applied to target code
- The majority of optimisations are performed either during or just after intermediate code generation, or during target code generation
  - source-level optimisations do not depend upon characteristics of the target machine and can be performed earlier
  - target-level optimisations depend upon the target architecture
    - sometimes an optimisation can consist of both

Area of Application

- Optimisations can be applied to different areas of a program
  - Local optimisations: those that are applied to ‘straight-line’ segments of code, i.e. with no jumps into or out of the sequence
    - easiest optimisations to perform
  - Global optimisations: those that extend beyond basic blocks but are confined to an individual procedure
    - more difficult to perform
  - Inter-procedural optimisations: those that extend beyond the boundaries of procedures to the entire program
    - most difficult optimisations to perform