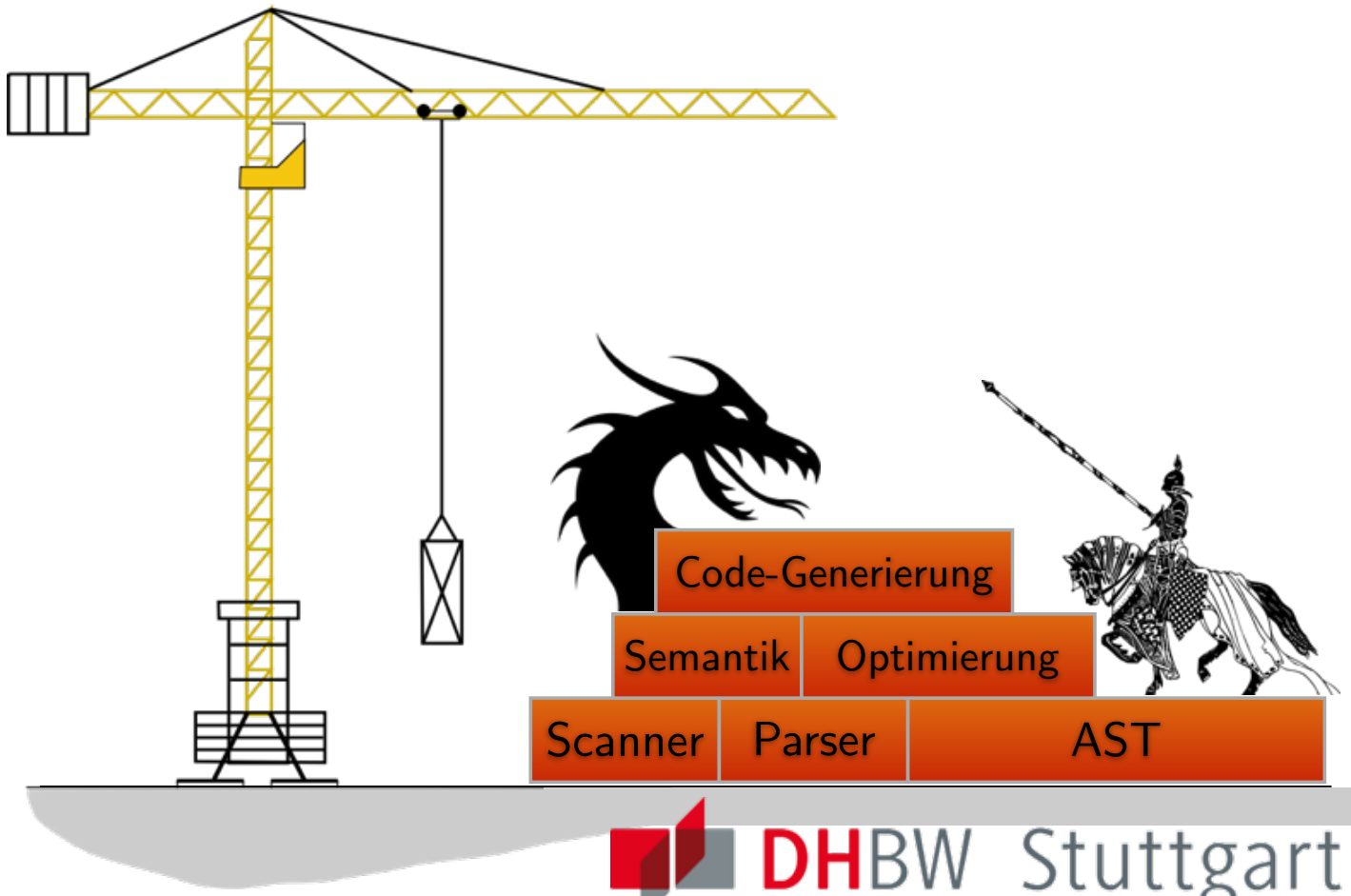
# Compilerbau

Stephan Schulz

stephan.schulz@dhbw-stuttgart.de



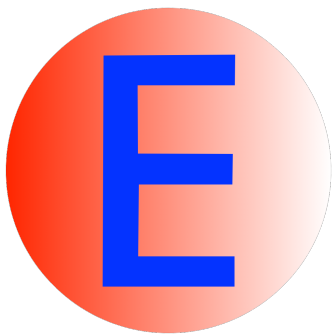

DHBW Stuttgart
Duale Hochschule
Baden-Württemberg

# Table of Contents

# Introduction

- Stephan Schulz
  - Dipl.-Inform., U. Kaiserslautern, 1995
  - Dr. rer. nat., TU München, 2000
  - Visiting professor, U. Miami, 2002
  - Visiting professor, U. West Indies, 2005
  - Visiting lecturer (Hildesheim, Offenburg, ...) since 2009
  - Industry experience: Building Air Traffic Control systems
    - System engineer, 2005
    - Project manager, 2007
    - Product Manager, 2013
  - Professor, DHBW Stuttgart, 2014

**Research: Logic & Deduction**



# This Course in Context

- *Formal languages and automata*
  - Basic theory - languages and automata
  - General grammars
  - Abstract parsing
  - Computability

Focus on foundations

- *Compiler construction*
  - Advanced theory - parsers and languages
  - Tools and their use
  - Writing parsers and scanners
  - Code generation and run time environment

Focus on practical applications

# Organization

- Lecture time: Wednesdays, 12:30-17:00
  - Lecture (with exercises): roughly 12:30-14:45
  - Lab: roughly 14:45-17:00
  - . . . but schedule will be flexible
  - Breaks as needed
  - No lecture on March 28th (*Second Conference on Artificial Intelligence and Theorem Proving*)
- Grading:
  - Lecture *Compilerbau*: Written Exam, grade averaged with *Formal Languages&Automata* for module grade
  - Lab: Pass/Fail based on success in exercises

# Computing Environment

- For practical exercises, you will need a complete Linux/UNIX environment. If you do not run one natively, there are several options:
  - You can install VirtualBox (`https://www.virtualbox.org`) and then install e.g. Ubuntu (`http://www.ubuntu.com/`) on a virtual machine. Make sure to install the *Guest Additions*
  - For Windows, you can install the complete UNIX emulation package Cygwin from `http://cygwin.com`
  - For MacOS, you can install `fink` (`http://www.finkproject.org/`) or MacPorts (`https://www.macports.org/`) and the necessary tools
- You will need at least `flex`, `bison`, `gcc`, `grep`, `sed`, `AWK`, `make`, and a good text editor

# Resources

- Course web page
  - `http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2018.html`
- Literature
  - Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*
  - Kenneth C. Louden: *Compiler Construction - Principles and Practice*
  - Ulrich Hedtstück: *Einführung in die theoretische Informatik*

# Exercise: Programming Languages

- Name and describe several modern programming languages!

# Modern Programming Languages

Desirable properties of high-level languages

- Expressive and flexible
  - Close to application domains
  - Good abstractions
  - Powerful constructs
  - Readable
- Compact
  - Programmer productivity depends on code length (!)
- Machine independent
  - Code should run on many platforms
  - Code should run on evolving platforms
- Strong error-checking
  - Static
  - Dynamic
- Efficiently executable

# Classification Criteria

- High-level vs. low level
- Type system
  - Statically typed (C, Pascal) vs. dynamically typed (Python, LISP)
  - Strongly typed (Pascal) vs. weakly typed (C, AWK)
  - Declaration of types (C++) vs. type inference (Haskell)
- Programming paradigm
  - Imperative vs. functional vs. logical
  - Unstructured (Basic) vs. procedural (Pascal, C) vs. object-oriented (Java, C++)
- Execution style
  - Interpreted (Basic, Logo, Scheme)
  - Compiled to abstract machine (Java, Prolog)
  - Compiled to native code (C, Modula-2)

Examples are typical, not strict

- Scheme can be compiled
- C can be interpreted
- Python can be used imperatively and has functional features
- ...

# Low-Level Code

- Machine code
  - Binary
  - Machine-specific
  - Operations (and operands) encoded in instruction words
  - Basic operations only
  - Manipulates finite number of registers
  - Direct access to memory locations
  - Flow control via conditional and unconditional jumps (think `goto`)
  - Basic data types (bytes, words)

Directly executable by processor

- Assembly languages
  - Textual representation of machine code
  - Symbolic names for operations and operands
  - Labels for addresses (code and data)

Direct one-to-one mapping to machine code

# Exercise: Low-Level Code – Minimal C

- Predefined global variables
  - Integers `R0, R1, R2, R3, R4`
  - Integer array `mem[MAXMEM]`
  - No new variables allowed
- No parameters (or return with value) for functions
- Flow control: Only `if` and `goto` (not `while`, `for`, ...)
  - No blocks after `if` (only one command allowed)
- Arithmetic only between `R0, R1, R2, R3, R4`
  - Result must be stored in one of `R0, R1, R2, R3, R4`
  - Operands: Only `R0, R1, R2, R3, R4` allowed (no nested sub-expressions)
  - Unary incement/decrement is ok (`R0++`)
  - `R0, R1, R2, R3, R4` can be stored in/loaded from `mem`, indexed with a fixed address or one of the variables.

# Exercise: Minimal C Example

```
/* Compute sum from 0 to R0, return result in R1 */

void user_code(void)
{
    /* R0 is the input value and limit */
    R1 = 0;    /* Sum, value returned */
    R2 = 0;    /* Loop counter */
    R3 = 1;    /* For increments */

loop:
    if(R2 > R0)
        goto end;
    R1 = R1+R2;
    R2 = R2+R3;
    goto loop;
end:
    return;
}
```

# Exercise: Low-Level Code

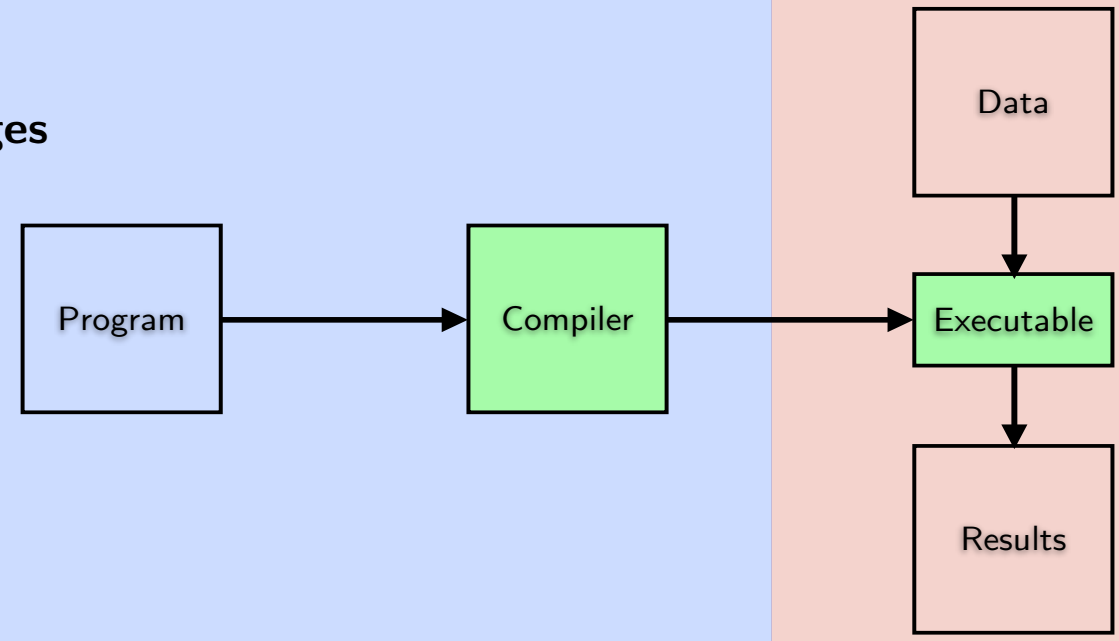
- Write (in Minimal C) the following functions:
  - A program computing the factorial of `R0`
  - A program computing the Fibonacci-number of `R0` iteratively
  - A program computing the Fibonacci-number of `R0` recursively
- You can find a frame for your code at the course web page, `http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2018.html`

# Surprise!

**Computers don't execute high-level languages (directly)!**

# Execution of high-level programs

**Compiled languages**




**Interpreted languages**

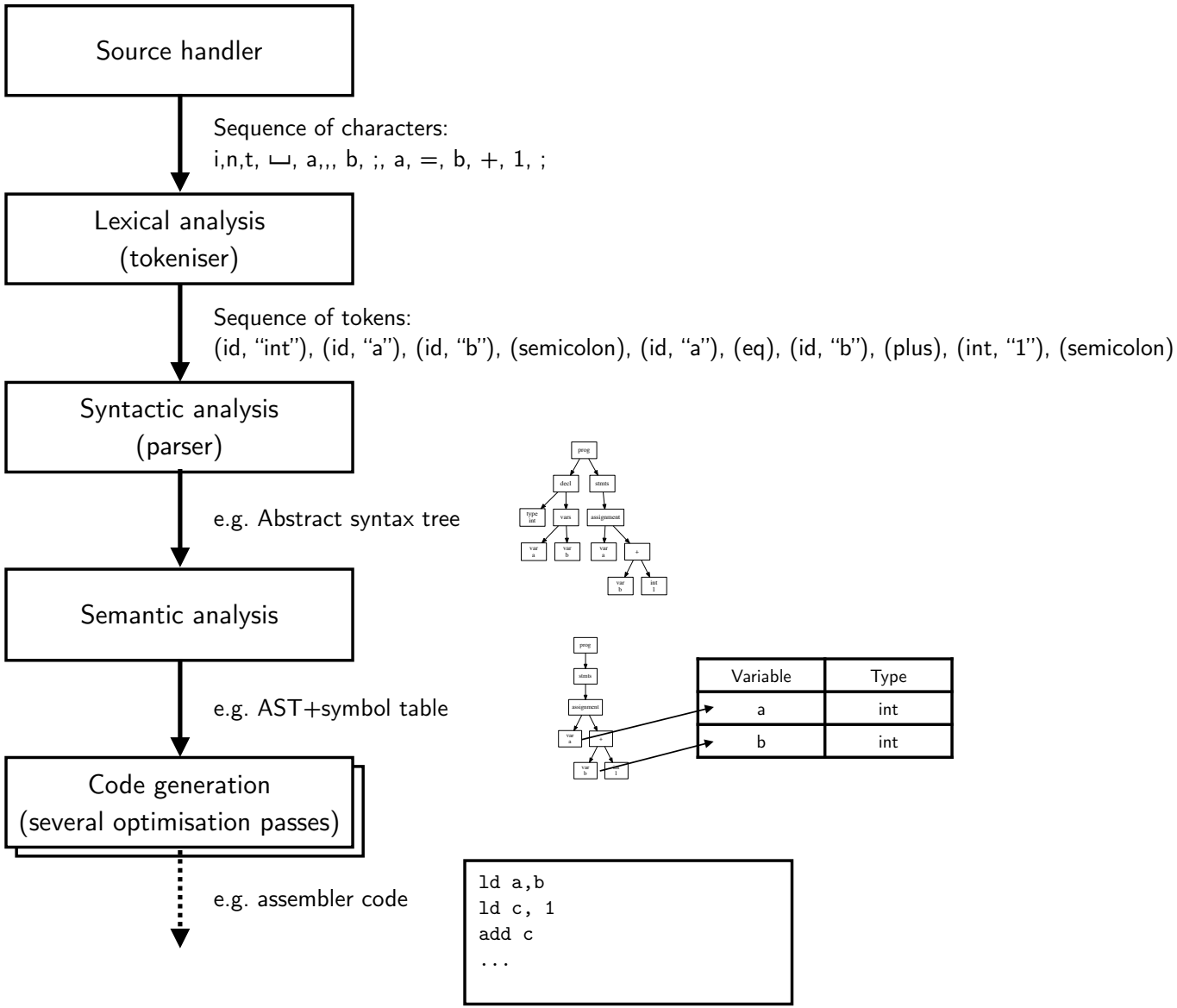
# Compilers

**Compilers translate high-level languages into low-level code!**

# Reminder: Syntactic Structure of Computer Languages

- Most computer languages are mostly context-free
  - Regular understory: vocabulary
    - Keywords, operators, identifiers
    - Described by regular expressions or regular grammar
    - Handled by (generated or hand-written) scanner/tokenizer/lexer
  - Context-free: program structure
    - Matching parenthesis, block structure, algebraic expressions, ...
    - Described by context-free gammar
    - Handled by (generated or hand-written) *parser*
  - Context-sensitive: e.g. declarations
    - Described by human-readable constraints
    - Handled in an ad-hoc fashion (e.g. symbol table)

# High-Level Architecture of a Compiler



# Source Handler

- Handles input files
- Provides character-by-character access
- May maintain file/line/colum (for error messages)
- May provide look-ahead
- May provide preprocessing (e.g. `include`)

**Result:** Sequence of characters (with positions)

- Nowadays often integrated with scanner

# Lexical Analysis/Scanning

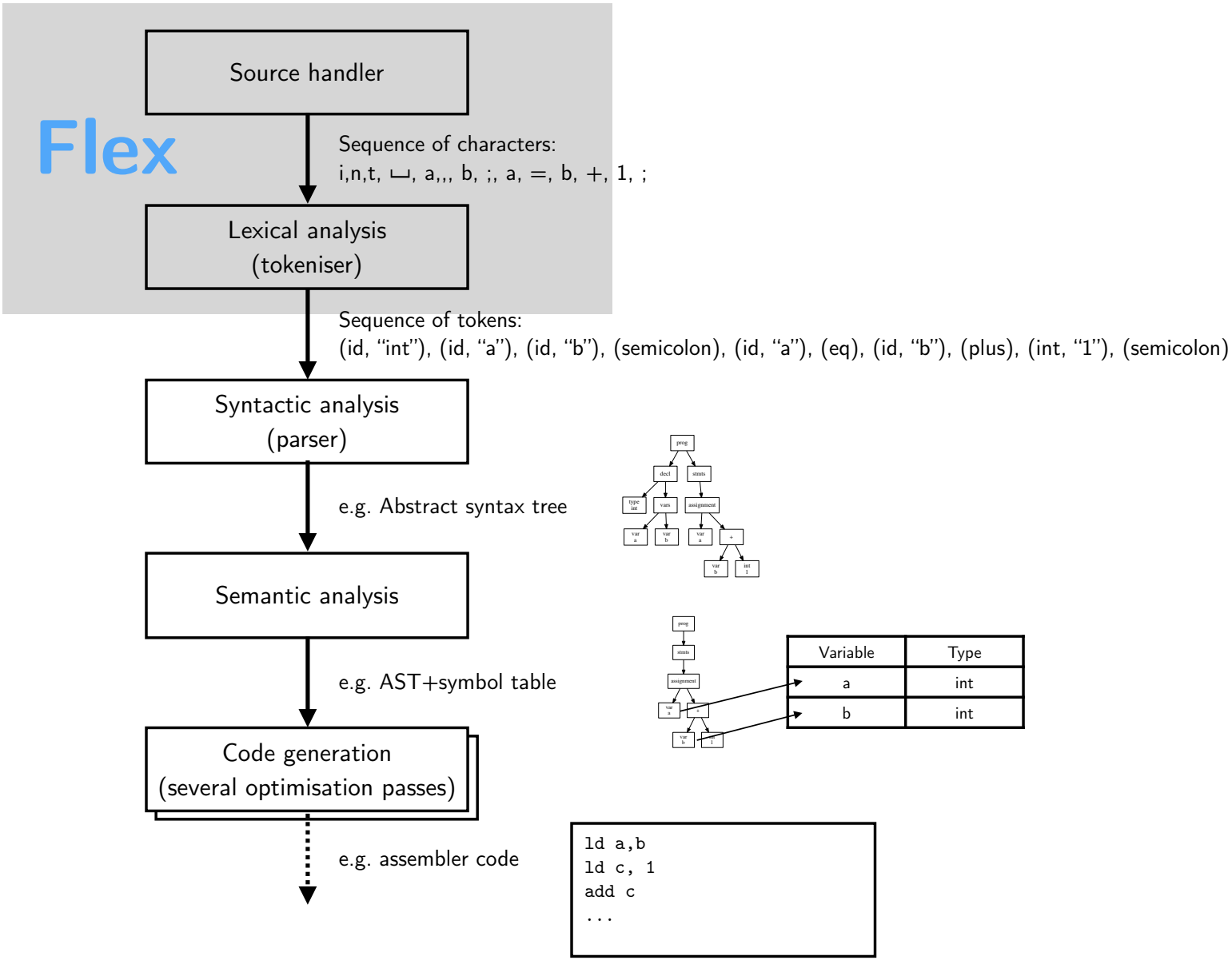
- Breaks program into token
- Typical tokens:
  - Reserved word (`if`, `while`)
  - Identifier (`i`, `counter`, `database`)
  - Numbers (`12`, `0x23`, `1.2e17`)
  - Symbols (`{`, `}`, `(`, `)`, `+`, `-`, `...`)

**Result:** Sequence of tokens

# Exercise: Lexical Analysis

```
int main(int argc, char* argv[])
{
    R0 = 0;
    R1 = 0;
    R2 = 0;
    R3 = 1;
    R4 = 1;
    for(int i = 0; i<MAXMEM; i++)
    {
        mem[i] = 0;
    }

    user_code();

    return 0;
}
```

# Automatisation with Flex

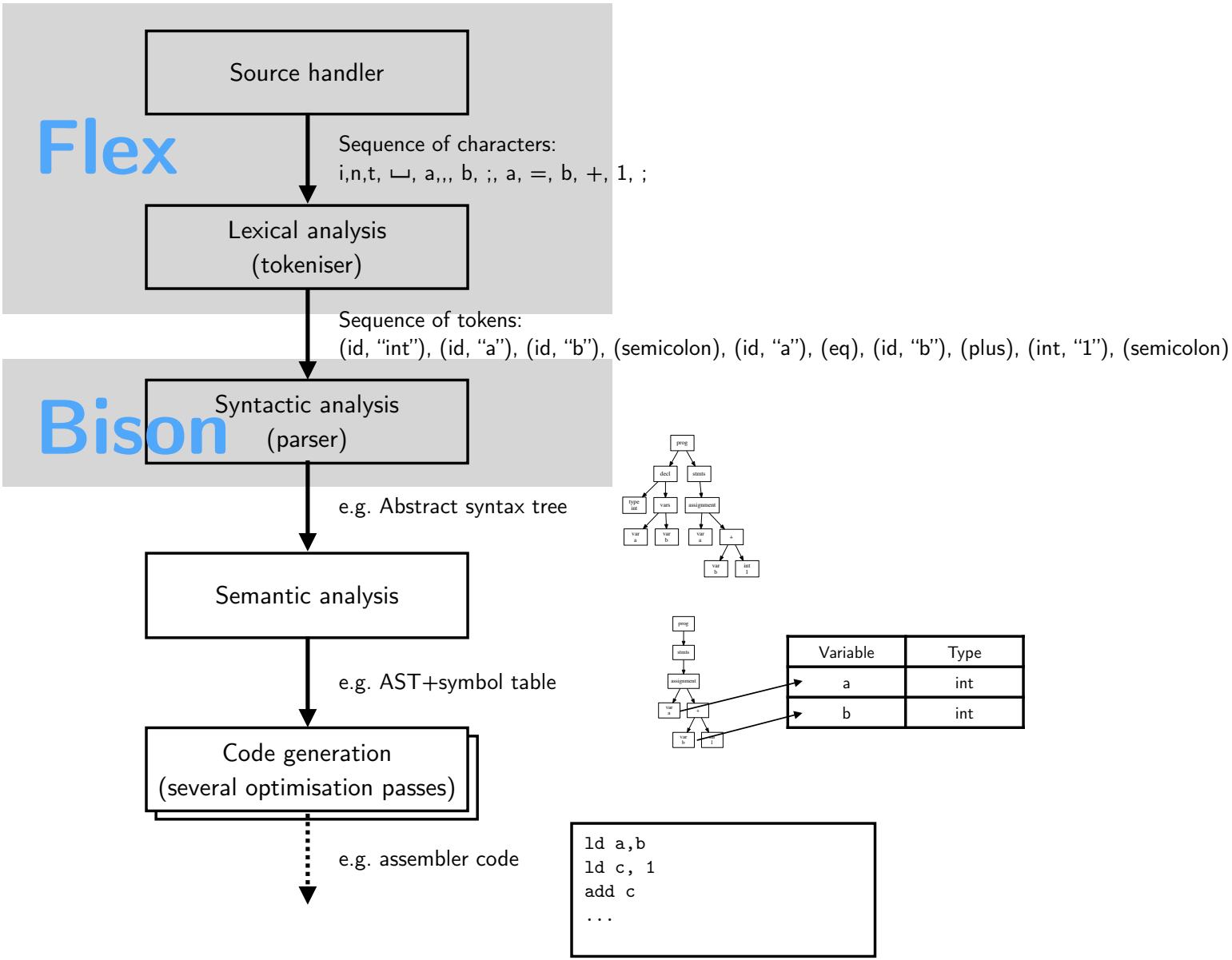


# Syntactical Analysis/Parsing

- Description of the language with a context-free grammar
- Parsing:
  - Try to build a *parse tree*/abstract syntax tree (AST)
  - Parse tree unambiguously describes structure of a program
  - AST reflects abstract syntax (can e.g. drop parenthesis)
- Methods:
  - Manual recursive descent parser
  - Automatic with a table-driven bottom-up parser

**Result:** Abstract Syntax Tree

# Automatisation with Bison



# Semantic Analysis

- Analyze static properties of the program
  - Which variable has which type?
  - Are all expressions well-typed?
  - Which names are defined?
  - Which names are referenced?
- Core tool: Symbol table

**Result:** Annotated AST

# Optimization

- Transform Abstract Syntax Tree to generate better code
  - Smaller
  - Faster
  - Both
- Mechanisms
  - Common sub-expression elimination
  - Loop unrolling
  - Dead code/data elimination
  - ...

**Result:** Optimized AST

# Code Generation

- Convert optimized AST into low-level code
- Target languages:
  - Assembly code
  - Machine code
  - VM code (z.B. JAVA byte-code, p-Code)
  - C (as a "portable assembler")
  - ...
- Also includes low-level/target-language dependent optimizations

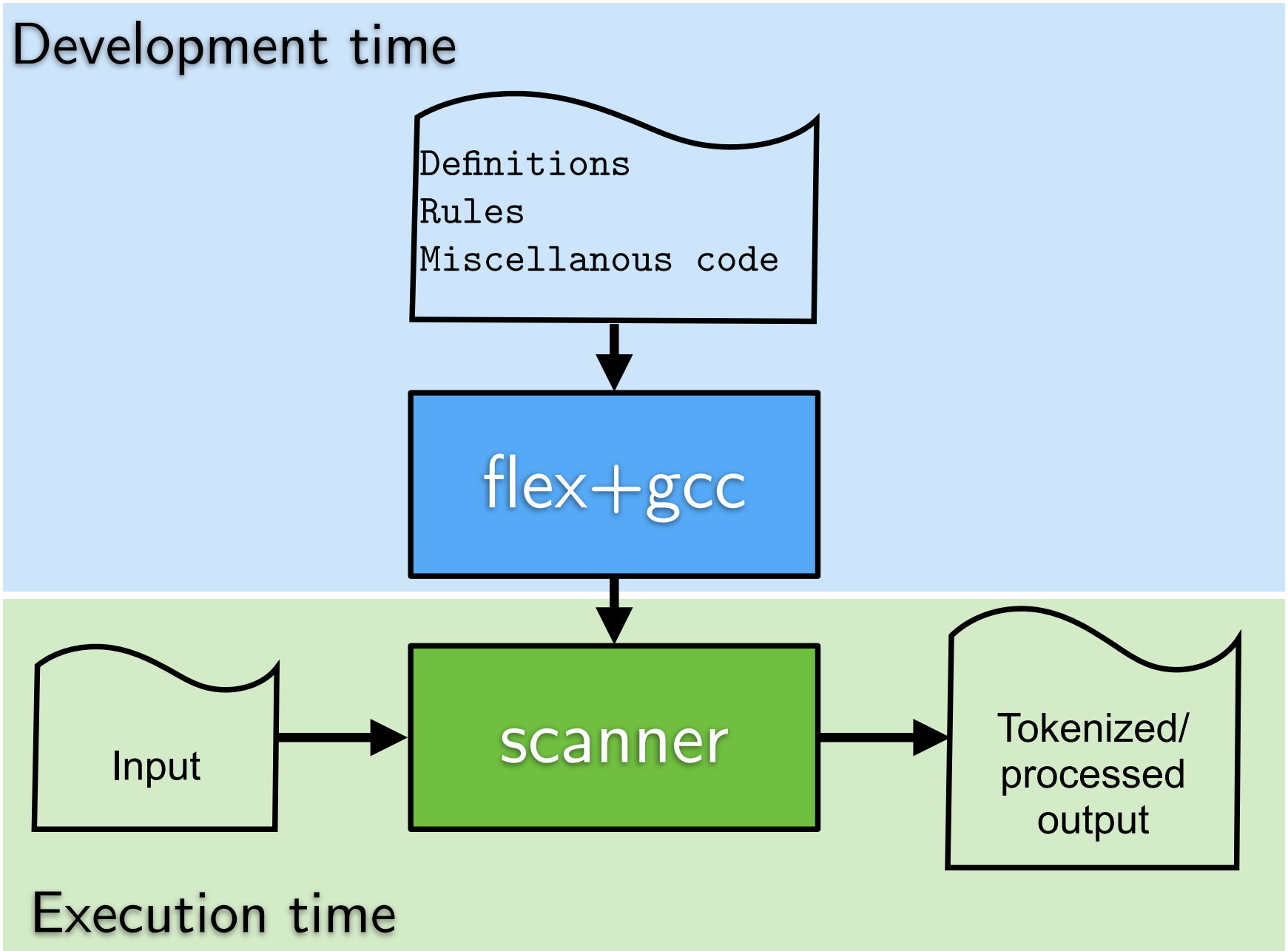
**Result:** Program in target language

# Refresher: Flex

# Flex Overview

- Flex is a scanner generator
- Input: Specification of a regular language and what to do with it
  - Definitions - named regular expressions
  - Rules - patters+actions
  - (miscellaneous support code)
- Output: Source code of scanner
  - Scans input for patterns
  - Executes associated actions
  - Default action: Copy input to output
  - Interface for higher-level processing: `yylex()` function

# Flex Overview



# Flex Example Task

- (Artificial) goal: Sum up all numbers in a file, separately for ints and floats
- Given: A file with numbers and commands
  - Ints: Non-empty sequences of digits
  - Floats: Non-empty sequences of digits, followed by decimal dot, followed by (potentially empty) sequence of digits
  - Command `print`: Print current sums
  - Command `reset`: Reset sums to 0.
- At end of file, print sums

# Flex Example Output

**Input**

```
12 3.1415
0.33333
print reset
2 11
1.5 2.5 print
1
print 1.0
```

**Output**

```
int:   12 ("12")
float: 3.141500 ("3.1415")
float: 0.333330 ("0.33333")
Current: 12 : 3.474830
Reset
int:   2 ("2")
int:   11 ("11")
float: 1.500000 ("1.5")
float: 2.500000 ("2.5")
Current: 13 : 4.000000
int:   1 ("1")
Current: 14 : 4.000000
float: 1.000000 ("1.0")
Final  14 : 5.000000
```

# Basic Structure of Flex Files

- Flex files have 3 sections
  - Definitions
  - Rules
  - User Code
- Sections are separated by %%
- Flex files traditionally use the suffix `.l`

# Example Code (definition section)

```
%option noyywrap

DIGIT    [0-9]

%{
   int    intval   = 0;
   double floatval = 0.0;
%}

%%
```

# Example Code (rule section)

```
{DIGIT}+    {
               printf( "int:   %d (\"%s\")\n", atoi(yytext), yytext );
               intval += atoi(yytext);
            }
{DIGIT}+"."{DIGIT}*        {
            printf( "float: %f (\"%s\")\n", atof(yytext),yytext );
            floatval += atof(yytext);
            }
reset {
        intval = 0;
        floatval = 0;
        printf("Reset\n");
      }
print {
         printf("Current: %d : %f\n", intval, floatval);
      }
\n|. {
        /* Skip */
      }
```

# Example Code (user code section)

```
%%
int main( int argc, char **argv )
{
   ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
      yyin = fopen( argv[0], "r" );
   else
      yyin = stdin;

   yylex();

   printf("Final  %d : %f\n", intval, floatval);
}
```

# Generating a scanner

```
> flex  -t numbers.l > numbers.c
> gcc   -c -o numbers.o numbers.c
> gcc numbers.o -o scan_numbers
> ./scan_numbers Numbers.txt
int:   12 ("12")
float: 3.141500 ("3.1415")
float: 0.333330 ("0.33333")
Current: 12 : 3.474830
Reset
int:   2 ("2")
int:   11 ("11")
float: 1.500000 ("1.5")
float: 2.500000 ("2.5")
...
```

# Flexing in detail

```
> flex -tv numbers.l > numbers.c
  scanner options: -tvI8 -Cem
  37/2000 NFA states
  18/1000 DFA states (50 words)
  5 rules
  Compressed tables always back-up
  1/40 start conditions
  20 epsilon states, 11 double epsilon states
  6/100 character classes needed 31/500 words
        of storage, 0 reused
  36 state/nextstate pairs created
  24/12 unique/duplicate transitions
  ...
  381 total table entries needed
```

# Exercise: Building a Scanner

- Download the `flex` example and input from `http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2018.html`
- Build and execute the program:
  - Generate the scanner with `flex`
  - Compile/link the C code with `gcc`
  - Execute the resulting program in the input file

# Definition Section

- Can contain `flex` options
- Can contain (C) initialization code
  - Typically `#include()` directives
  - Global variable definitions
  - Macros and type definitions
  - Initialization code is embedded in %{ and %}
- Can contain definitions of regular expressions
  - Format: `NAME RE`
  - Defined NAMES can be referenced later

# Example Code (definition section) (revisited)

```
%%option noyywrap

DIGIT    [0-9]

%{
   int    intval   = 0;
   double floatval = 0.0;
%}


%%
```

# Rule Section

- This is the core of the scanner!
- Rules have the form `PATTERN ACTION`
- Patterns are regular expressions
  - Typically use previous definitions
- THERE IS WHITE SPACE BETWEEN PATTERN AND ACTION!
- Actions are C code
  - Can be embedded in { and }
  - Can be simple C statements
  - For a token-by-token scanner, must include `return` statement
  - Inside the action, the variable `yytext` contains the text matched by the pattern
  - Otherwise: Full input file is processed

# Example Code (rule section) (revisited)

```
{DIGIT}+    {
               printf( "int:   %d (\"%s\")\n", atoi(yytext), yytext );
               intval += atoi(yytext);
            }
{DIGIT}+"."{DIGIT}*        {
            printf( "float: %f (\"%s\")\n", atof(yytext),yytext );
            floatval += atof(yytext);
            }
reset {
        intval = 0;
        floatval = 0;
        printf("Reset\n");
      }
print {
         printf("Current: %d : %f\n", intval, floatval);
      }
\n|. {
        /* Skip */
      }
```

# User code section

- Can contain all kinds of code
- For stand-alone scanner: must include `main()`
- In `main()`, the function `yylex()` will invoke the scanner
- `yylex()` will read data from the file pointer `yyin` (so `main()` must set it up reasonably

# Example Code (user code section) (revisited)

```
%%
int main( int argc, char **argv )
{
   ++argv, --argc;  /* skip over program name */
   if ( argc > 0 )
      yyin = fopen( argv[0], "r" );
   else
      yyin = stdin;

   yylex();

   printf("Final  %d : %f\n", intval, floatval);
}
```

# A comment on comments

- Comments in Flex are complicated
  - . . . because nearly everything can be a pattern
- Simple rules:
  - Use old-style C comments `/* This is a comment */`
  - Never start them in the first column
  - Comments are copied into the generated code
  - Read the manual if you want the dirty details

# Flex Miscellany

- Flex online:
  - `https://github.com/westes/flex`
  - Manual: `https://westes.github.io/flex/manual/`
  - REs: `https://westes.github.io/flex/manual/Patterns.html`
- `make` knows `flex`
  - Make will automatically generate `file.o` from `file.l`
  - Be sure to set `LEX=flex` to enable `flex` extensions
  - `Makefile` example:

```
LEX=flex
all: scan_numbers
numbers.o: numbers.l

scan_numbers: numbers.o
        gcc numbers.o -o scan_numbers
```

# Parsing mit Bison

# YACC/Bison

- Yacc - Yet Another Compiler Compiler
  - Originally written ≈1971 by Stephen C. Johnson at AT&T
  - LALR parser generator
  - Translates grammar into syntax analyzer

- GNU Bison
  - Written by Robert Corbett in 1988
  - Yacc-compatibility by Richard Stallman
  - Output languages now C, C++, Java
- Yacc, Bison, BYacc, . . . mostly compatible (POSIX P1003.2)

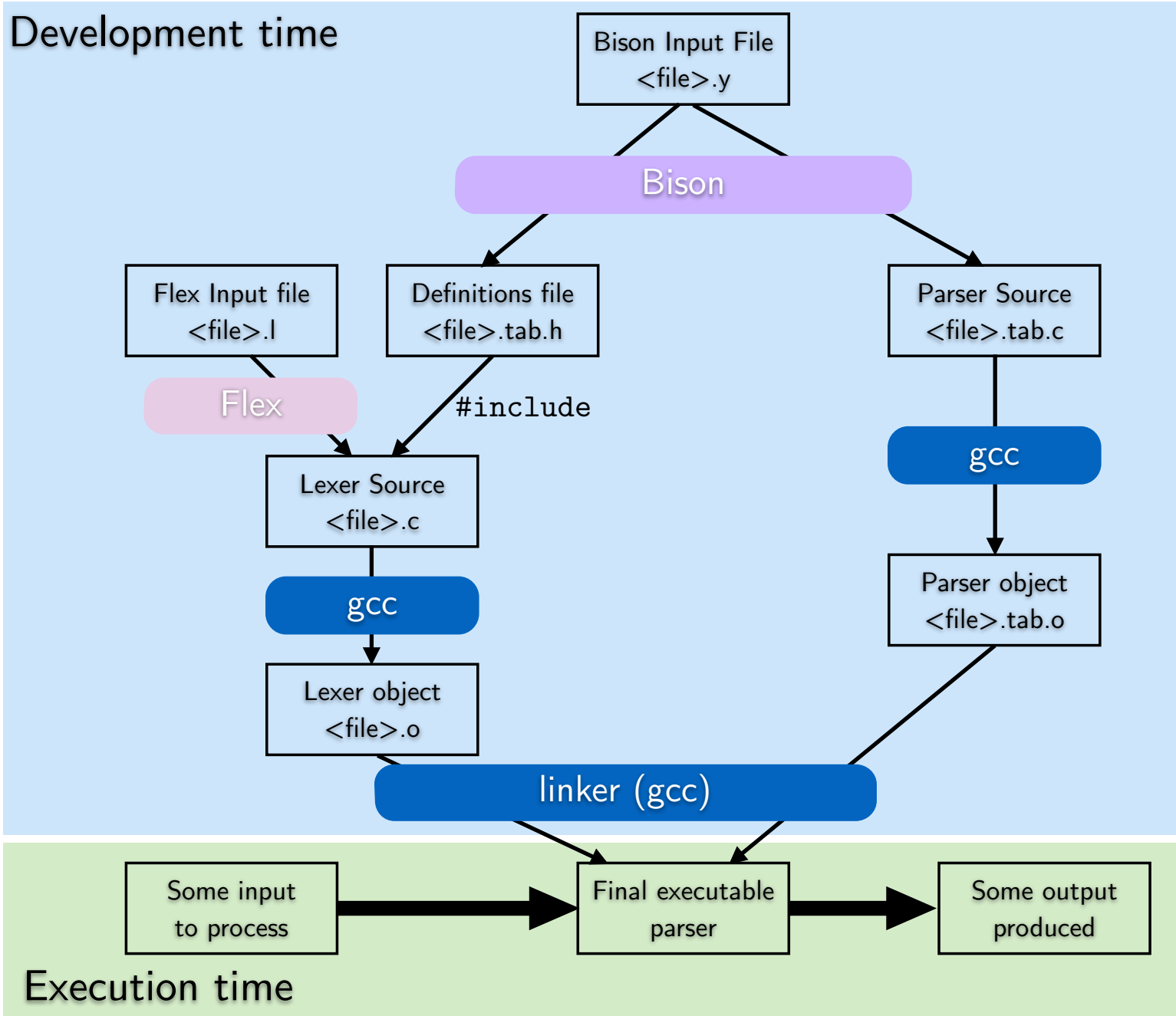
# Yacc/Bison Background

- By default, Bison constructs a 1 token Look-Ahead Left-to-right Rightmost-derivation or LALR(1) parser
  - Input tokens are processed left-to-right
  - Shift-reduce parser:
    - Stack holds tokens (terminals) and non-terminals
    - Tokens are shifted from input to stack. If the top of the stack contains symbols that represent the right hand side (RHS) of a grammar rule, the content is reduced to the LHS
    - Since input is reduced left-to-right, this corresponds to a rightmost derivation
    - Ambiguities are solved via look-ahead and special rules
    - If input can be reduced to start symbol, success!
    - Error otherwise
- LALR(1) is efficient in time and memory
  - Can parse "all reasonable languages"
  - For unreasonable languages, Bison (but not Yacc) can also construct GLR (General LR) parsers
    - Try all possibilities with back-tracking
    - Corresponds to the *non-determinism* of stack machines

# Yacc/Bison Overview

- Bison reads a specification file and converts it into (C) code of a parser
- Specification file: Definitions, grammar rules with actions, support code
  - Definitions: Token names, associated values, includes, declarations
  - Grammar rules: Non-terminal with alternatives, action associated with each alternative
  - Support code: e.g. `main()` function, error handling...
  - Syntax similar to (F)lex
    - Sections separated by %%
    - Special commands start with %
- Bison generates function `yyparse()`
- Bison needs function `yylex()`
  - Usually provided via (F)lex

# Yacc/Bison workflow



# Example task: Desk calculator

- Desk calculator
  - Reads algebraic expressions and assignments
  - Prints result of expressions
  - Can store values in registers `R0-R99`
- Example session:

```
[Shell] ./scicalc
R10=3*(5+4)
> RegVal: 27.000000
(3.1415*R10+3)
> 87.820500
R9=(3.1415*R10+3)
> RegVal: 87.820500
R9+R10
> 114.820500
...
```

# Abstract grammar for desk calculator (partial)

$G_{DC} = \langle V_N, V_T, P, S \rangle$

- $V_T = \{$`PLUS`, `MULT`, `ASSIGN`, `OPENPAR`, `CLOSEPAR`, `REGISTER`, `FLOAT`, $\ldots\}$
  - Some terminals are single characters (+, =, $\ldots$)
  - Others are complex: `R10`, `1.3e7`
  - Terminals ("tokens") are generated by the lexer
- $V_N = \{$`stmt`, `assign`, `expr`, `term`, `factor`, $\ldots\}$

- $P$ :

```
stmt    →  assign
        |  expr
assign  →  REGISTER ASSIGN expr
expr    →  expr PLUS term
        |  term
term    →  term MULT factor
        |  factor
factor  →  REGISTER
        |  FLOAT
        |  OPENPAR expr CLOSEPAR
```

- $S$ = *handwave*
  - For a single statement, $S =$ `stmt`
  - In practice, we need to handle sequences of statements and empty input lines (not reflected in the grammar)

# Parsing statements (1)

- Example string: R10 = (4.5+3*7)
- Tokenized: `REGISTER ASSIGN OPENPAR FLOAT PLUS FLOAT MULT FLOAT CLOSEPAR`
  - In the following abbreviated R, A, O, F, P, F, M, F, C
- Parsing state:
  - Unread input (left column)
  - Current stack (middle column, top element on right)
  - How state was reached (right column)
- Parsing:

| Input | Stack | Comment |
|---|---|---|
| `R A O F P F M F C` | | Start |
| `A O F P F M F C` | `R` | Shift R to stack |
| `O F P F M F C` | `R A` | LA: Shift A to stack |
| `F P F M F C` | `R A O` | Shift O to stack |
| `P F M F C` | `R A O F` | Shift F to stack |
| `P F M F C` | `R A O factor` | Reduce F |
| `P F M F C` | `R A O term` | Reduce `factor` |

...

# Parsing statements (2)

| | | |
|---|---|---|
| R A O F P F M F C | | Start |
| A O F P F M F C | R | Shift R to stack |
| O F P F M F C | R A | LA: Shift A to stack |
| F P F M F C | R A O | Shift O to stack |
| P F M F C | R A O F | Shift F to stack |
| P F M F C | R A O factor | Reduce F |
| P F M F C | R A O term | Reduce factor |
| P F M F C | R A O expr | LA: Reduce term |
| F M F C | R A O expr P | Shift P |
| M F C | R A O expr P F | Shift F |
| M F C | R A O expr P factor | Reduce F |
| M F C | R A O expr P term | Reduce factor |
| F C | R A O expr P term M | LA: Shift M |
| C | R A O expr P term M F | Shift F |
| C | R A O expr P term M factor | Reduce F |
| C | R A O expr P term | Reduce tMf |
| C | R A O expr | Reduce ePt |
| | R A O expr C | Shift C |
| | R A factor | Reduce OeC |
| | R A term | Reduce factor |
| | R A expr | Reduce term |
| | stmt | Reduce RAe |

# Lexer interface

- Bison parser requires `yylex()` function
- `yylex()` returns token
  - Token text is defined by regular expression pattern
  - Tokens are encoded as integers
  - Symbolic names for tokens are defined by Bison in generated header file
    - By convention: Token names are all `CAPITALS`
- `yylex()` provides optional semantic value of token
  - Stored in global variable `yylval`
  - Type of `yylval` defined by Bison in generated header file
    - Default is `int`
    - For more complex situations often a `union`
    - For our example: Union of double (for floating point values) and integer (for register numbers)

# Lexer for desk calculator (1)

```
/*
     Lexer for a minimal "scientific" calculator.

     Copyright 2014 by Stephan Schulz, schulz@eprover.org.

     This code is released under the GNU General Public Licence
     Version 2.
 */

%option noyywrap

%{
   #include "scicalcparse.tab.h"
%}
```

# Lexer for desk calculator (2)

```
DIGIT       [0-9]
INT         {DIGIT}+
PLAINFLOAT {INT}|{INT}[.]|{INT}[.]{INT}|[.]{INT}
EXP         [eE](\+|-)?{INT}
NUMBER      {PLAINFLOAT}{EXP}?
REG         R{DIGIT}{DIGIT}?

%%

"*" {return MULT;}
"+" {return PLUS;}
"=" {return ASSIGN;}
"(" {return OPENPAR;}
")" {return CLOSEPAR;}
\n  {return NEWLINE;}
```

# Lexer for desk calculator (3)

```
{REG}    {
            yylval.regno = atoi(yytext+1);
            return REGISTER;
         }

{NUMBER} {
            yylval.val = atof(yytext);
            return FLOAT;
        }

[ ] { /* Skip  whitespace*/ }

 /* Everything else is an invalid character. */
.    { return ERROR;}

%%
```

# Data model of desk calculator

- Desk calculator has simple state
  - 100 floating point registers
  - `R0`-`R99`
- Represented in C as array of doubles:

```
#define MAXREGS 100

double regfile[MAXREGS];
```

- Needs to be initialized in support code!

# Bison code for desk calculator: Header

```
%{
   #include <stdio.h>

   #define MAXREGS 100

   double regfile[MAXREGS];

   extern int yyerror(char* err);
   extern int yylex(void);
%}

%union {
   double val;
   int    regno;
}
```

# Bison code for desk calculator: Tokens

```
%start stmtseq

%left  PLUS
%left  MULT
%token ASSIGN
%token OPENPAR
%token CLOSEPAR
%token NEWLINE
%token REGISTER
%token FLOAT
%token ERROR

%%
```

# Actions in Bison

- Bison is based on syntax rules with associated actions
  - Whenever a reduce is performed, the action associated with the rule is executed
- Actions can be arbitrary C code
- Frequent: semantic actions
  - The action sets a semantic value based on the semantic value of the symbols reduced by the rule
  - For terminal symbols: Semantic value is `yylval` from Flex
  - Semantic actions have "historically valuable" syntax
    - Value of reduced symbol: `$$`
    - Value of first symbol in syntax rule body: `$1`
    - Value of second symbol in syntax rule body: `$2`
    - ...
    - Access to named components: `$<val>1`

# Bison code for desk calculator: Grammar (1)

```
stmtseq: /* Empty */
    | NEWLINE stmtseq       {}
    | stmt  NEWLINE stmtseq {}
    | error NEWLINE stmtseq {};  /* After an error,
                                    start afresh */
```

- Head: sequence of statements
- First body line: Skip empty lines
- Second body line: separate current statement from rest
- Third body line: After parse error, start again with new line

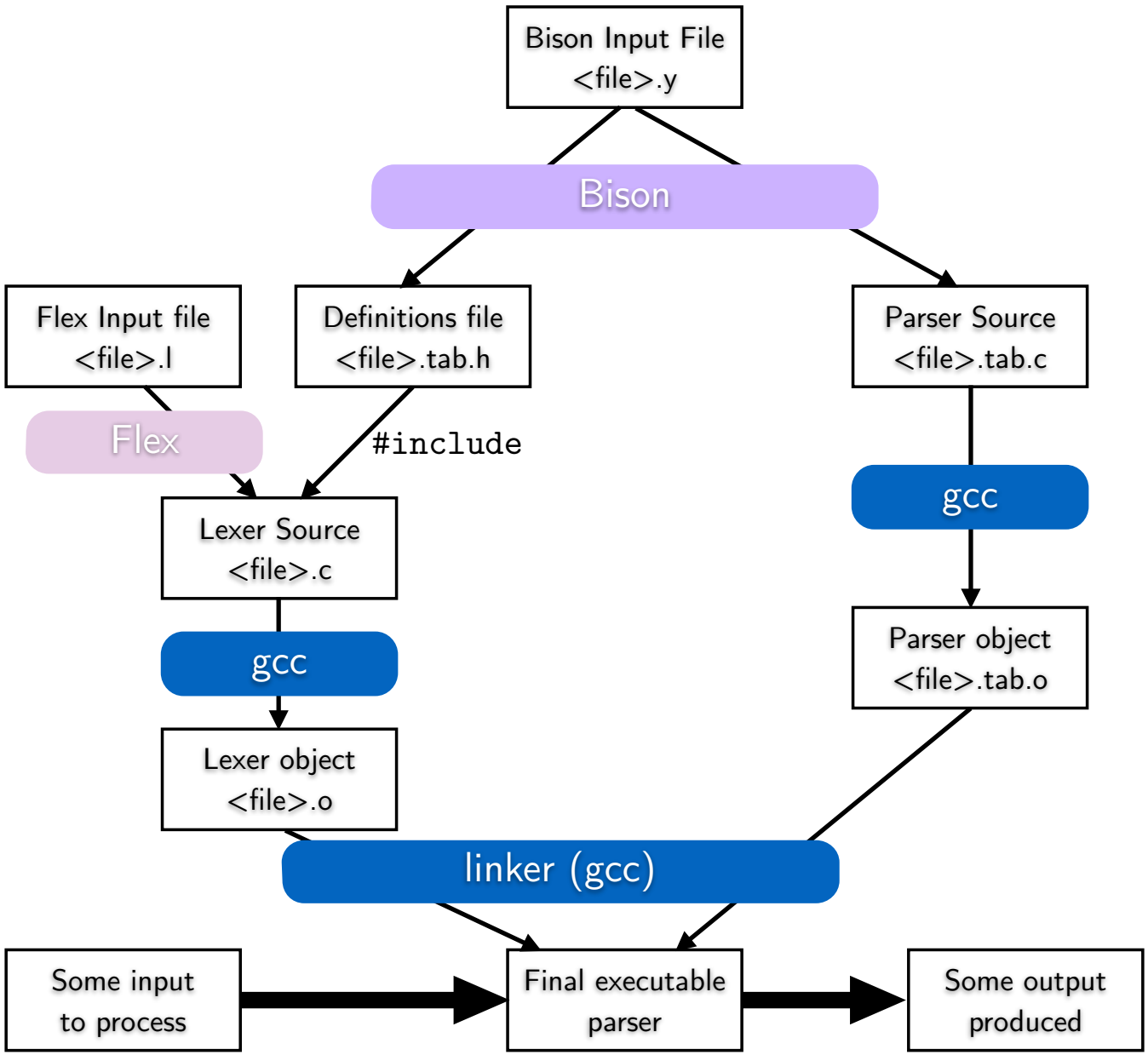
# Bison code for desk calculator: Grammar (2)

```
stmt: assign  {printf("> RegVal: %f\n", $<val>1);}
     |expr    {printf("> %f\n", $<val>1);};

assign: REGISTER ASSIGN expr {regfile[$<regno>1] = $<val>3;
                              $<val>$ = $<val>3;} ;

expr: expr PLUS term {$<val>$ = $<val>1 + $<val>3;}
    | term {$<val>$ =  $<val>1;};

term: term MULT factor {$<val>$ = $<val>1 * $<val>3;}
    | factor {$<val>$ =  $<val>1;};

factor: REGISTER {$<val>$ = regfile[$<regno>1];}
      | FLOAT  {$<val>$ =  $<val>1;}
      | OPENPAR expr CLOSEPAR {$<val>$ =  $<val>2;};
```

# Bison code for desk calculator: Support code

```
int yyerror(char* err)
{
   printf("Error: %s\n", err);
   return 0;
}

int main (int argc, char* argv[])
{
  int i;

  for(i=0; i<MAXREGS; i++)
  {
     regfile[i] = 0.0;
  }
  return yyparse();
}
```

# Reminder: Workflow and dependencies



# Building the calculator

1. Generate parser C code and include file for lexer
   - `bison -d scicalcparse.y`
   - Generates `scicalcparse.tab.c` and `scicalcparse.tab.h`
2. Generate lexer C code
   - `flex -t scicalclex.l > scicalclex.c`
3. Compile lexer
   - `gcc -c -o scicalclex.o scicalclex.c`
4. Compile parser and support code
   - `gcc -c -o scicalcparse.tab.o scicalcparse.tab.c`
5. Link everything
   - `gcc scicalclex.o scicalcparse.tab.o -o scicalc`
6. Fun!
   - `./scicalc`

End Lecture 1

# Refresher: Grammars

# Formal Grammars: Motivation

Formal grammars describe formal languages!

- Deriviative approach
  - A grammar has a set of rules
  - Rules replace words with words
  - A word that can be derived from a special start symbol is in the language of the grammar

**In the concrete case of programming languages, "Words of the language" are syntactically correct programs!**

# Grammars: Examples

$S \to aA, \quad A \to bB, \quad B \to \varepsilon$
generates $ab$ (starting from $S$): $S \to aA \to abB \to ab$

$S \to \varepsilon, \quad S \to aSb$
generates $a^n b^n$

# Grammars: definition

Noam Chomsky defined a grammar as a quadruple

$$G = \langle V_N, V_T, P, S \rangle \tag{1}$$

with

1. the set of non-terminal symbols $V_N$,
2. the set of terminal symbols $V_T$,
3. the set of production rules $P$ of the form

$$\alpha \to \beta \tag{2}$$

$$\text{with} \quad \alpha \in V^* V_N V^*, \beta \in V^*, V = V_N \cup V_T$$

4. the distinguished start symbol $S \in V_N$.

# Grammars: Shorthand

For the sake of simplicity, we will be using the short form

$$\alpha \to \beta_1 | \cdots | \beta_n \quad \text{instead of} \quad \begin{array}{l} \alpha \to \beta_1 \\ \vdots \\ \alpha \to \beta_n \end{array} \tag{3}$$

# Example: C identifiers

We want to define a grammar

$$G = \langle V_N, V_T, P, S \rangle \tag{4}$$

to describe identifiers of the `C` programming language:

- alpha-numeric words
- which must not start with a digit
- and may contain an underscore (_)

$V_N = \{I, R, L, D\}$ (identifier, rest, letter, digit),

$V_T = \{\texttt{a}, \cdots, \texttt{z}, \texttt{A}, \cdots, \texttt{Z}, \texttt{0}, \cdots, \texttt{9}, \_\}$,

$P = \{$

$$\begin{aligned} I &\rightarrow LR|\_R|L|\_ \\ R &\rightarrow LR|DR|\_R|L|D|\_ \\ L &\rightarrow \texttt{a}|\cdots|\texttt{z}|\texttt{A}|\cdots|\texttt{Z} \\ D &\rightarrow \texttt{0}|\cdots|\texttt{9}\} \end{aligned}$$

$S = I$.

# Formal grammars: derivation

Derivation: description of operation of grammars
Given the grammar

$$G = \langle V_N, V_T, P, S \rangle, \tag{5}$$

we define the relation

$$x \Rightarrow_G y \tag{6}$$

$$\text{iff } \exists u, v, p, q \in V^* : (x = upv) \wedge (p \to q \in P) \wedge (y = uqv) \tag{7}$$

pronounced as "$G$ derives $y$ from $x$ in one step".
We also define the relation

$$x \Rightarrow_G^* y \text{ iff } \exists w_0, \ldots, w_n \tag{8}$$

$$\text{with } w_0 = x, w_n = y, w_{i-1} \Rightarrow_G w_i \text{ for } i \in \{1, \cdots, n\}$$

pronounced as "$G$ derives $y$ from $x$ (in zero or more steps)".

# Formal grammars: derivation example I

$$G = \langle V_N, V_T, P, S \rangle \tag{9}$$

with

1. $V_N = \{S\}$,
2. $V_T = \{0\}$,
3. $P = \{S \to 0S, \quad S \to 0\}$,
4. $S = S$.

Derivations of $G$ have the general form

$$S \Rightarrow 0S \Rightarrow 00S \Rightarrow \cdots \Rightarrow 0^{n-1}S \Rightarrow 0^n \tag{10}$$

Thus, the language produced by $G$ (or the language of G) is

$$L(G) = \{0^n | n \in \mathbb{N}; n > 0\}. \tag{11}$$

# Formal grammars: derivation example II

$$G = \langle V_N, V_T, P, S \rangle \tag{12}$$

with

1. $V_N = \{S\}$,
2. $V_T = \{0, 1\}$,
3. $P = \{S \to 0S1, \quad S \to 01\}$,
4. $S = S$.

Derivations of $G$ have the general form

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow \cdots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n1^n. \tag{13}$$

The language of $G$ is

$$L(G) = \{0^n1^n | n \in \mathbb{N}; n > 0\}. \tag{14}$$

**Reminder: L(G) is not regular!**

# The Chomsky hierarchy (0)

**Definition (Grammar of type 0)**

Every Chomsky grammar $G = (N, \Sigma, P, S)$ is of Type 0 or unrestricted.

# The Chomsky hierarchy (1)

## Definition (context-sensitive grammar)

A grammar $G = (N, \Sigma, P, S)$ is of is Type 1 (context-sensitive) if all productions are of the form

$$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2 \text{ with } A \in N; \alpha_1, \alpha_2 \in V^*, \beta \in VV^*$$

where $V = N \cup \Sigma$.
Exception: the rule $S \to \varepsilon$ is allowed if $S$ does not appear on the right-hand side of any rule

- The context must not be modified.
- Rules never derive shorter words
  - except for the empty word in the first step
- In fact, every grammar without contracting rules (monotonic grammar) can be rewritten as a context-sensitive grammar.

# The Chomsky hierarchy (2)

## Definition (context-free grammar)

A grammar $G = (N, \Sigma, P, S)$ is of is Type 2 (context-free) if all productions are of the form

$$A \to \beta \text{ with } A \in N; \beta \in V^*$$

- Only single non-terminals are replaced
  - independent of their context
- Contracting rules are allowed!
  - context-free grammars are not a subset of context-sensitive grammars
  - but: context-free languages are a subset of context-sensitive languages
  - reason: contracting rules can be removed from context-free grammars, but not from context-sensitive ones

# The Chomsky hierarchy (3)

## Definition (right-linear grammar)

A grammar $G = (N, \Sigma, P, S)$ is of Type 3 (right-linear or regular) if all productions are of the form

$$A \to aB$$

$$\text{with } A \in N; B \in N \cup \{\epsilon\}; a \in \Sigma \cup \{\epsilon\}$$

- only one NTS on the left
- on the right: one TS, one NTS (in that order), both, or neither
- analogy with automata is obvious

# Formal grammars and formal languages

**Definition (language classes)**

A language is called

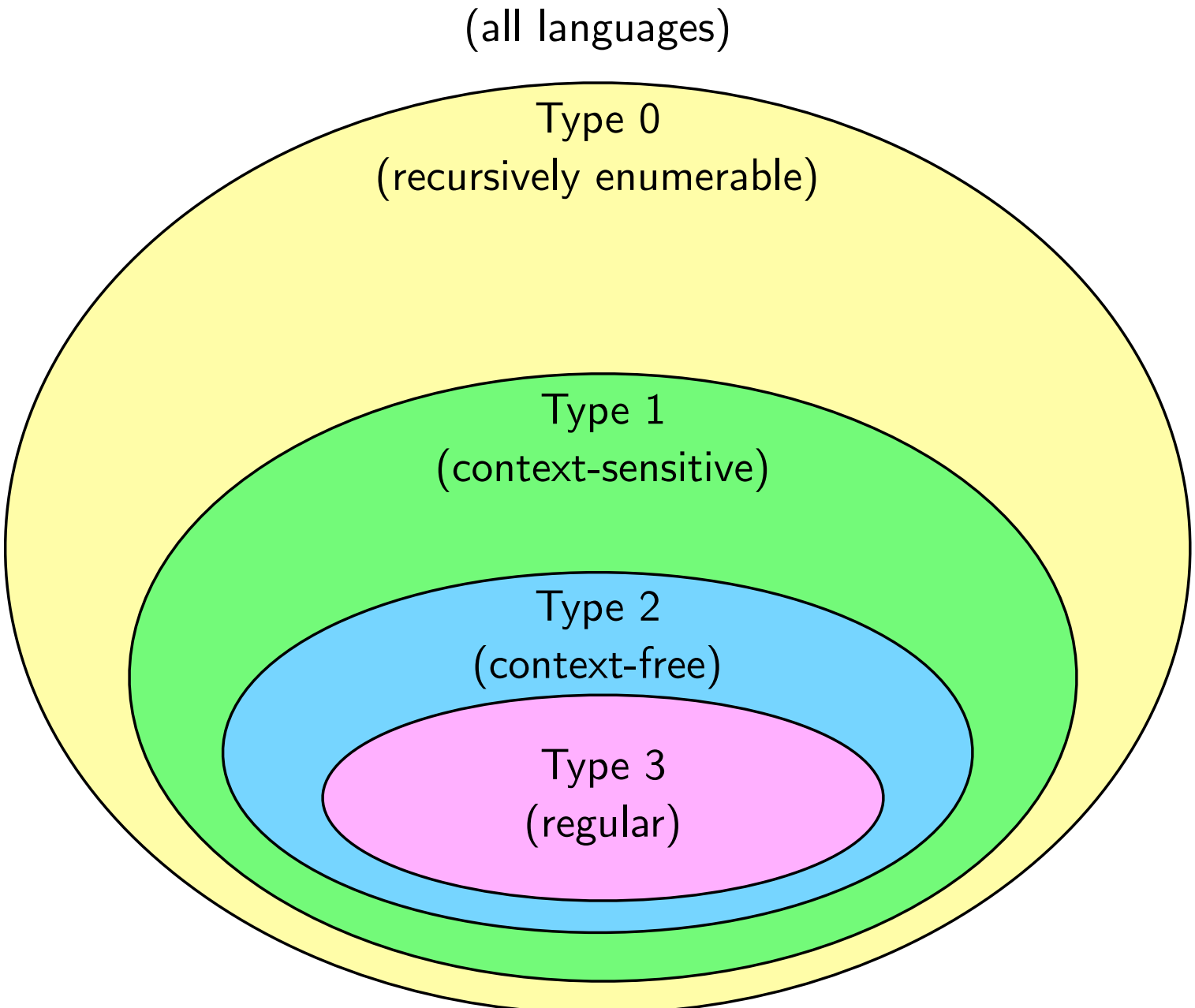
recursively enumerable, context-sensitive, context-free, or regular,

if it can be generated by a

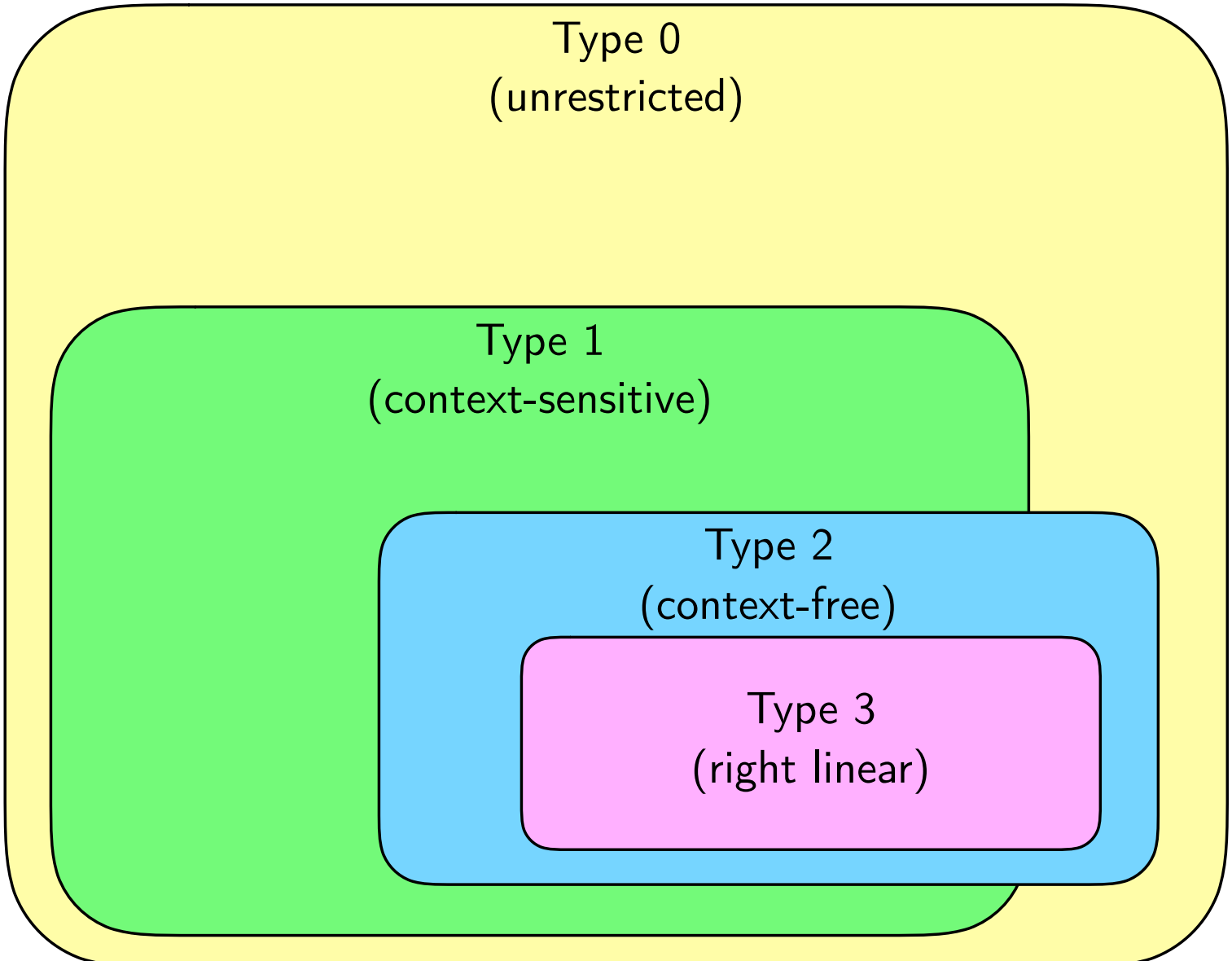
unrestricted, context-sensitive, context-free, or regular

grammar, respectively.

# The Chomsky Hierarchy for Languages



# The Chomsky Hierarchy for Grammars



# The Chomsky hierarchy: exercises

$$G = \langle V_N, V_T, P, S \rangle \tag{15}$$

with

1. $V_N = \{S, A, B\}$,
2. $V_T = \{0\}$,
3. $P$ :

$$\begin{aligned} S &\to \varepsilon && 1 \\ S &\to ABA && 2 \\ AB &\to 00 && 3 \\ 0A &\to 000A && 4 \\ A &\to 0 && 5 \end{aligned}$$

4. $S = S$.

a) What is $G$'s highest type?

b) Show how $G$ derives the word 00000.

c) Formally describe the language $L(G)$.

d) Define a regular grammar $G'$ equivalent to $G$.

# The Chomsky hierarchy: exercises (cont.)

An octal constant is a finite sequence of digits starting with 0 followed by at least one digit ranging from 0 to 7. Define a regular grammar encoding exactly the set of possible octal constants.

# Context-free grammars

- Reminder: $G = \langle V_N, V_T, P, S \rangle$ is context-free, if all $l \to r \in P$ are of the form $A \to \beta$ with
  - $A \in V_N$ and $\beta \in V^*$
- Context-free languages/grammars are highly relevant
  - Core of most programming languages
  - Algebraic expressions
  - XML
  - Many aspects of human language

# Grammars in Practice

- Most programming languages are described by context-free grammars (with extra "semantic" constraints)
- Grammars generate languages
- PDAs and e.g. CYK-Parsing recognize words
- For compiler we need to ...
  - identify correct programs
  - and understand their structure!

# Lexing and Parsing

- Lexer: Breaks programs into tokens/lexemes
  - Smallest parts with semantic meaning
    - Strictly: Lexeme is the text, token is the type
    - . . . but few people are strict!
  - Can be recognized by regular languages/patterns
  - Example: `1`, `2`, `5` are all Integers
  - Example: `i`, `handle`, `stream` are all Identifiers
  - Example: `>`, `>=`, `*` are all individual operators
- Parser: Recognizes program structure
  - Language described by a grammar that has token types as terminals, not individual characters
  - Parser builds *parse tree*

# Introduction: nanoLang

# Our first language: *nanoLang*

- Simple but Turing-complete language
- Block-structured
  - Functions with parameters
  - Blocks of statements with local variables
- Syntax "C-like" but simplified
  - Basic flow control (`if`, `while`, `return`)
- Simple static type system
  - Integers (64 bit signed)
  - Strings (immutable)

# *nanoLang* "Hello World"

```
# The first ever nanoLang program

Integer main()
{
    print "Hello World\n";
    return 0;
}
```

# More Substantial *nanoLang* Example

```
Integer hello(Integer repeat, String message)
{
    Integer i;
    i = 0;
    while(i<repeat)
    {
        print message;
        i = i+1;
    }
    return 0;
}

Integer main()
{
    hello(10, "Hello\n");
    return 0;
}
```

# *nanoLang* Lexical Structure

- Reserved words:
  - `if`, `while`, `return`, `print`, `else`, `Integer`, `String`
- Comments: # to the end of the line
- Variable length tokens:
  - Identifier (letter, followed by letters and digits)
  - Strings (enclosed in double quotes (¨`This is a string`¨)
  - Integer numbers (non-empty sequences of digits)
- Other tokens:
  - Brackets: `(`,`)`, {,}
  - Operators: `+`, `-`, `*`, `/`
  - Comparison operators: `>`, `>=`, `<`, `<=`, `!=`
  - Equal sign = (used for comparison and assignments!)
  - Separators: `,`, `;`

# *nanoLang* Program Structure

- A *nanoLang* program consists of a number of definitions
  - Definitions can define global variables or functions
  - All symbols defined in the global scope are visible everywhere in the global scope
- Functions accept arguments and return values
  - Functions consist of a header and a statement blocks
  - Local variables can be defined in statement blocks
- Statements:
  - `if` and `if/else`: Conditional execution
  - `while`: Structured loops
  - `return`: Return value from function
  - `print`: Print value to Screen
  - Assignment: Set variables to values
  - Function calls (return value ignored)
- Expressions:
  - Integers: Variables, numbers, `+`,`-`,`*`,`/`
  - Booleans: Compare two values of equal type

# Exercise: Fibonacci in *nanoLang*

- Write a recursive and an iterative implementation of Fibonacci numbers in *nanoLang*

# *nanoLang* Grammar (Bison format) (0 -tokens)

```
%start prog

%token IDENT
%token STRINGLIT
%token INTLIT
%token INTEGER STRING
%token IF WHILE RETURN PRINT ELSE
%token EQ NEQ LT GT LEQ GEQ
%left  PLUS MINUS
%left  MULT DIV
%right UMINUS
%token OPENPAR CLOSEPAR
%token SEMICOLON COMA
%token OPENCURLY CLOSECURLY

%token ERROR
```

# *nanoLang* Grammar (Bison format) (1)

```
prog: /* Nothing */
    | prog def
;

def: vardef
   | fundef
;

vardef: type idlist SEMICOLON
;

idlist: IDENT
      | idlist COMA IDENT

fundef: type IDENT OPENPAR params CLOSEPAR body
;
```

# *nanoLang* Grammar (Bison format) (2)

```
type: STRING
    | INTEGER
;

params: /* empty */
    | paramlist
;

paramlist: param
         | paramlist COMA param
;

param: type IDENT
;

body: OPENCURLY vardefs stmts CLOSECURLY
;
```

# *nanoLang* Grammar (Bison format) (3)

```
vardefs: /* empty */
       | vardefs vardef
;

stmts: /* empty */
     | stmts stmt
;

stmt: while_stmt
    | if_stmt
    | ret_stmt
    | print_stmt
    | assign
    | funcall_stmt
;
```

# *nanoLang* Grammar (Bison format) (4)

```
while_stmt: WHILE OPENPAR boolexpr CLOSEPAR body
;

if_stmt: IF OPENPAR boolexpr CLOSEPAR body
       | IF OPENPAR boolexpr CLOSEPAR body ELSE body
;

ret_stmt: RETURN expr SEMICOLON
;
```

# *nanoLang* Grammar (Bison format) (5)

```
print_stmt: PRINT expr SEMICOLON
;

assign: IDENT EQ expr SEMICOLON
;

funcall_stmt: funcall SEMICOLON
;

boolexpr: expr EQ expr
        | expr NEQ expr
        | expr LT expr
        | expr GT expr
        | expr LEQ expr
        | expr GEQ expr
;
```

# *nanoLang* Grammar (Bison format) (6)

```
expr: funcall
    | INTLIT
    | IDENT
    | STRINGLIT
    | OPENPAR expr CLOSEPAR
    | expr PLUS expr
    | expr MINUS expr
    | expr MULT expr
    | expr DIV expr
    | MINUS expr
;
```

# *nanoLang* Grammar (Bison format) (7)

```
funcall: IDENT OPENPAR args CLOSEPAR
;

args: /* empty */
    | arglist
;

arglist: expr
       | arglist COMA expr
;
```

End Lecture 2

# Words vs. Derivations vs. Structure

- Assume a grammar $G = \langle V_N, V_T, P, S \rangle$ with associated language $L(G)$
- Question so far: $w \in L(G)$?
  - I.e. $S \Rightarrow_G^* w$?
- Now we are interested in the *structure* of words
  - $a + b * c + d$ should be interpreted as $(a + (b * c)) + d$
  - We are looking at *derivations* that reflect this structure
  - We are looking for grammars that support such derivations (and only such derivations)

# A Running Example

- We will consider the set of well-formed expressions over $x, +, *, (, )$ as an example, i.e. the language $L(G)$ for $G$ as follows:
  - $V_N = \{E\}$
  - $V_T = \{(, ), +, *, x\}$
  - Start symbol is $E$
  - Productions:
    1. $E \to x$
    2. $E \to (E)$
    3. $E \to E + E$
    4. $E \to E * E$

# Derivations

## Definition (Derivation)

Assume a Grammar $G$. A derivation of a word $w_n$ in $L(G)$ is a sequence $S \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_n$ where $S$ is the start symbol, and each $w_i$ is generated from its predecessor by application of a production of the grammar.

- Example: Consider our running example. We bold the replaced symbol. The following is a derivation of $x + x + x * x$:

$$
\begin{aligned}
& \mathbf{E} \\
\Longrightarrow \quad & \mathbf{E} + E \\
\Longrightarrow \quad & E + E + \mathbf{E} \\
\Longrightarrow \quad & \mathbf{E} + E + E * E \\
\Longrightarrow \quad & x + \mathbf{E} + E * E \\
\Longrightarrow \quad & x + x + \mathbf{E} * E \\
\Longrightarrow \quad & x + x + x * \mathbf{E} \\
\Longrightarrow \quad & x + x + x * x
\end{aligned}
$$

# Rightmost/Leftmost Derivations

## Definition (Rightmost/leftmost derivation)

- A derivation is called a rightmost derivation, if at any step it replaces the rightmost non-terminal in the current word.
- A derivation is called a leftmost derivation, if at any step it replaces the leftmost non-terminal in the current word.

- Examples:
  - The derivation on the previous slide is neither leftmost nor rightmost.
  - $\mathbf{E} \Longrightarrow E + \mathbf{E} \Longrightarrow E + E + \mathbf{E} \Longrightarrow E + E + E * \mathbf{E} \Longrightarrow E + E + \mathbf{E} * x \Longrightarrow E + \mathbf{E} + x * x \Longrightarrow \mathbf{E} + x + x * x \Longrightarrow x + x + x * x$ is a rightmost derivation.

# Parse trees

## Definition (Parse tree)

A parse tree for a derivation in a grammar $G = \langle V_N, V_T, P, S \rangle$ is an ordered, labelled tree with the following properties:

- Each node is labelled with a symbol from $V_N \cup V_T$
- The root of the tree is labelled with the start symbol $S$.
- Each inner node is labelled with a single non-terminal symbol from $V_N$
- If an inner node with label $A$ has children labelled with symbols $\alpha_1, \ldots, \alpha_n$, then there is a production $A \to \alpha_1 \ldots \alpha_n$ in $P$.

+

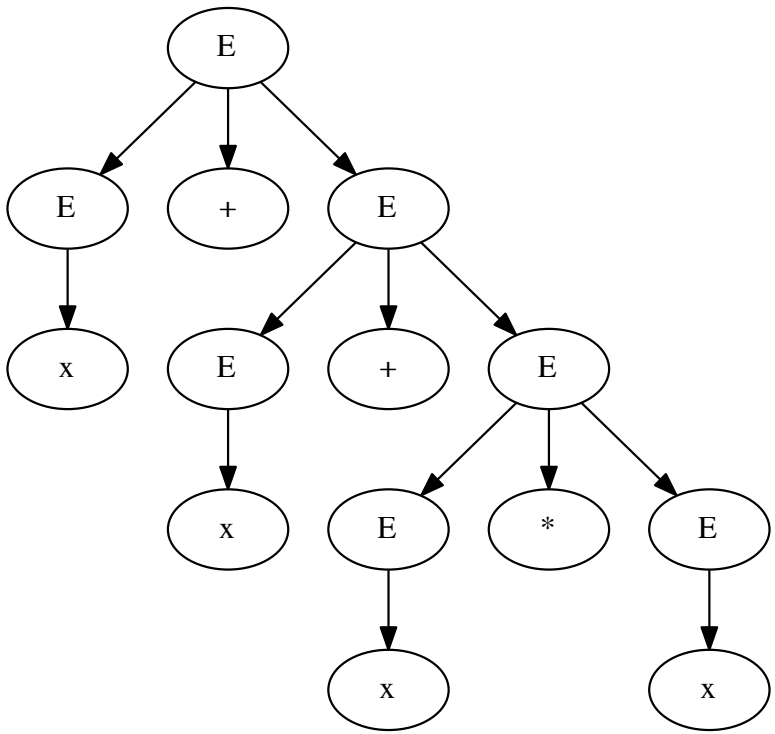
- The parse tree represents a derivation of the word formed by the labels of the leaf nodes
- It abstracts from the order in which productions are applied.

# Parse trees: Example

Consider the following derivation:

$$\mathbf{E} \Longrightarrow E + \mathbf{E} \Longrightarrow E + E + \mathbf{E} \Longrightarrow E + E + E * \mathbf{E} \Longrightarrow$$
$$E + E + \mathbf{E} * x \Longrightarrow E + \mathbf{E} + x * x \Longrightarrow \mathbf{E} + x + x * x \Longrightarrow x + x + x * x$$

It can be represented by a sequence of parse trees:

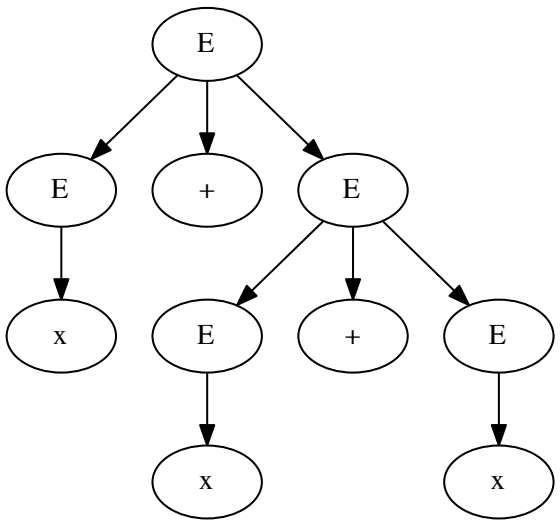


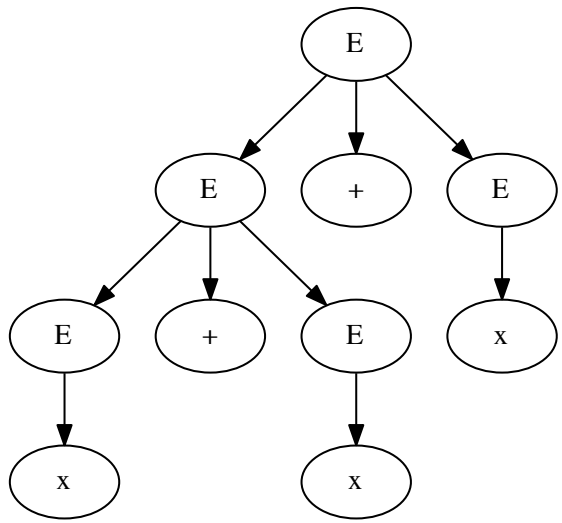
# Ambiguity

## Definition (Ambiguity)

A grammar $G = \langle V_N, V_T, P, S \rangle$ is ambiguous, if it has multiple different parse trees for a word $w$ in $L(G)$.

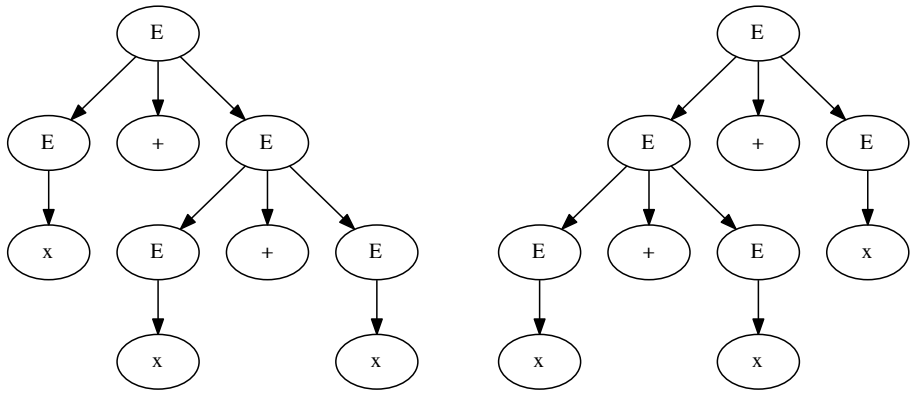
- Consider our running example with the following productions:
  1. $E \to x$
  2. $E \to (E)$
  3. $E \to E + E$
  4. $E \to E * E$
- The following 2 parse trees represent derivations of $x + x + x$:



# Exercise: Ambiguity is worse. . .

- Consider our example and the parse trees from the previous slide:
  1. $E \to x$
  2. $E \to (E)$
  3. $E \to E + E$
  4. $E \to E * E$




- Provide a rightmost derivation for the right tree.
- Provide a rightmost derivation for the left tree.
- Provide a leftmost derivation for the left tree.
- Provide a leftmost derivation for the right tree.

# Exercise: Eliminating Ambiguity

- Consider our running example with the following productions:
  1. $E \to x$
  2. $E \to (E)$
  3. $E \to E + E$
  4. $E \to E * E$
- Define a grammar $G'$ with $L(G) = L(G')$ that is not ambiguous, that respects that $*$ has a higher precedence than $+$, and that respects left-associativity for all operators.
- Hints:
  - The different precedence levels (products and sums) need to be generated from different non-terminals
  - Within each level, symmetry needs to be broken (one term vs. the rest) in a way reflecting left-associativity

Solution

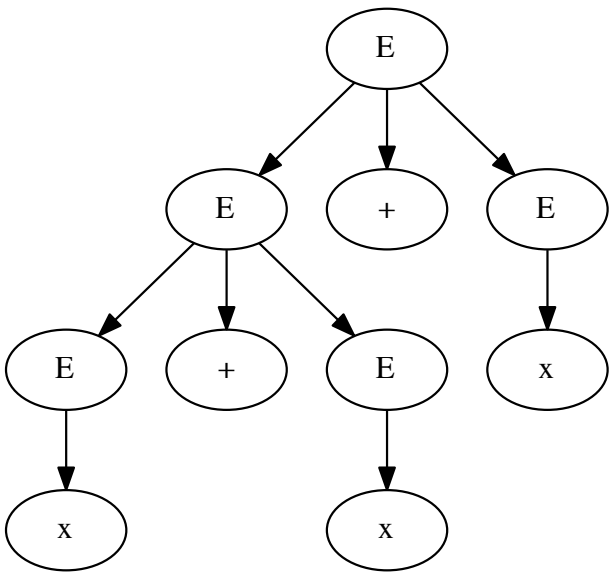
# Abstract Syntax Trees (1)

- Abstract Syntax Trees represent the structure of a derivation without the specific details
- Think: "Parse trees without the syntactic sugar"
  - Not an unambiguous definition
  - Many different versions exist
  - Important: Do not lose *relevant* information!
- I use the following:
  - Unnecessary terminal symbols are dropped
  - When possible, the terminal defining the type of a subtree is merged with the non-terminal parent node

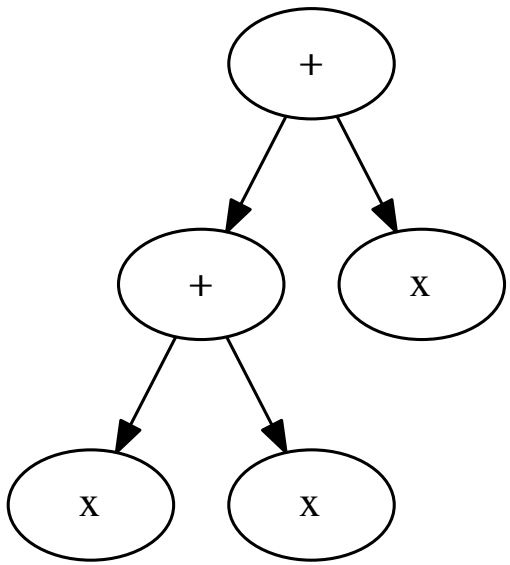
# Abstract Syntax Trees (2)

- Abstract Syntax Trees represent the structure of a derivation without the specific details
- Example:

Parse Tree:




Corresponding AST:



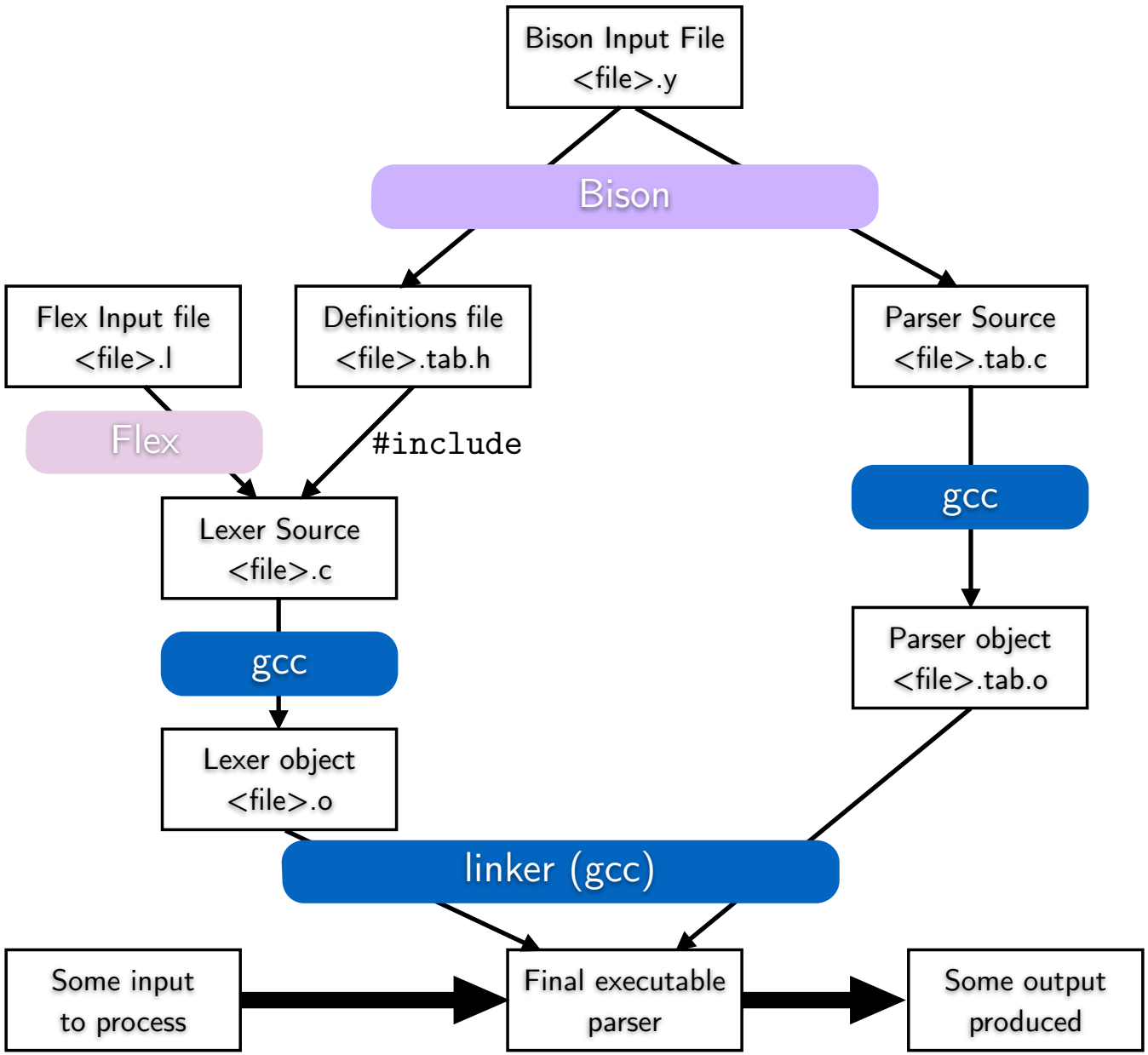
# Exercise: Abstract Syntax Trees

- Define two non-trivial arithmetic expressions (according to our running example)
  - Each should have at least 4 operators
  - At least one should use two pairs of parentheses
- Find a rightmost derivation in $G$ that respects associativity and operator precedence for each expression
- Find the corresponding parse tree for each expression
- Convert the parse trees to abstract syntax trees

# Building Complex Software

- Complex projects consist of many diffent files
  - Some user-edited, some generated
  - E.g. `flex` input file
  - E.g. C source files
  - E.g. object files
  - E.g. final executable
- Any given generated file may depend on a number of input files
  - If one of the input files changes, the generated file needs to be updated
  - This can cascade (assume A depends on B, B depends on C, C depends on D - what happens if D changes?)

# Example: Flex und Bison Workflow



# Build Automation with `make`

- `make` is the traditional UNIX tool for build automation
  - Models dependencies
  - Triggers rebuilds when necessary
- Important concepts:
  - Rules, consisting of
    - Targets – that need to be rebuild
    - Dependencies – inputs required for building a target
    - Commands – to create the target from the dependencies
  - Implicit and built-in rules
    - `Make` knows how to build a `.o` from a `.c`
    - . . . and many other recipes
    - Users can write generic rules to exend this knowledge
  - Makefile variables
    - . . . are used to instanciate generic rules
    - E.g. name of the C compiler
    - E.g. compiler options
    - E.g. name of `lex`

# Makefiles

- Build instructions are encoded in a `makefile`
  - `make` looks for `Makefile` or `makefile`
  - Optional: `make -f myfunnymakefilename`
- Syntax of make files
  - Setting of variables: `CC=gcc`
  - Targets and dependencies: `target: dep1 dep2 dep3`
  - Recipes: All lines after a target that start with a Tab character describe commands to be executed to build the target
- `make` always builds the *first* target in the Makefile
  - If there are multiple targets, use a *phony* pseudotarget that depends on all real targets
  - Make computes all dependencies and determines the build order
  - For each target it checks if the file exists and is newer than all its dependencies
    - If not, the recipe is executed to update the target
    - If target is not a file, it is rebuild every time

# Makefile Example

```
LEX  = flex
YACC = bison
CC   = gcc
LD   = gcc

all: scicalc

scicalclex.c: scicalclex.l scicalcparse.tab.h

scicalcparse.tab.h: scicalcparse.y
	$(YACC) -d scicalcparse.y

scicalcparse.tab.c: scicalcparse.y
	$(YACC) -d scicalcparse.y

scicalcparse.tab.o: scicalcparse.tab.c

scicalc: scicalclex.o scicalcparse.tab.o
	$(LD) scicalclex.o scicalcparse.tab.o -o scicalc
```

# A Second Example

```
LEX  = flex
YACC = bison
CC   = gcc
LD   = gcc
CFLAGS = -ggdb

all: nanolex nanolex_sa

nanolex_sa.c: nanolex_sa.l

nanolex_sa: nanolex_sa.c
        gcc -o nanolex_sa nanolex_sa.c

nanolex: nanolex.c
        gcc -o nanolex nanolex.c
```

# More on Make

- Standard `make` for modern UNIX dialects is GNU Make
  - Documentation: `https://www.gnu.org/software/make/manual/`

Use of make is optional, but highly recommended

# Flex/Bison Interface

- Bison calls the function `yylex` to get the next token
- `yylex` executes user rules (pattern/action)
  - User actions return token (integer value)
  - Additionally: `yylval` can be set and is available in Bison via the `$$/$1/...` mechanism
- `yylval` provides the *semantic value* of a token
  - For complex languages: Use a (pointer to) a struct
    - Content: Position, string values, numerical values, ...
  - Type of `yylval` if set in *Bison* file!

    ```
    % define api.value.type {YourType}
    ```
- It's probably a good idea to have a relatively simple type for `yylval`

End Lecture 3

# Context: Where are we?

Front End

Source handler

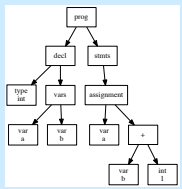
Sequence of characters:
i,n,t, ␣, a,,, b, ;, a, =, b, +, 1, ;

Lexical analysis
(tokeniser)

Sequence of tokens:
(id, "int"), (id, "a"), (id, "b"), (semicolon), (id, "a"), (eq), (id, "b"), (plus), (int, "1"), (semicolon)

Syntactic analysis
(parser)

e.g. Abstract syntax tree




Semantic analysis

e.g. AST+symbol table+attributes


| Variable | Type |
|---|---|
| a | int |
| b | int |

Code generation
(several optimisation passes)

e.g. assembler code

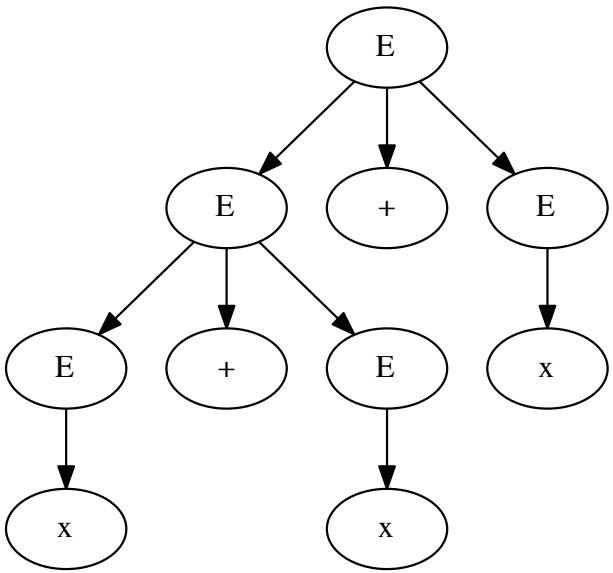
```
ld a,b
ld c, 1
add c
...
```

Back End

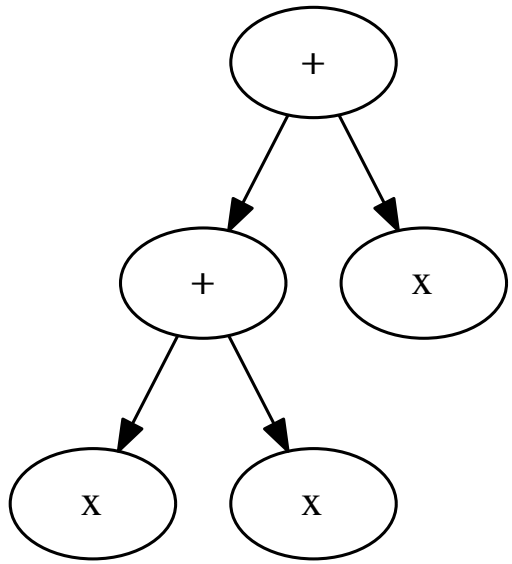
# Reminder: Abstract Syntax Trees

- Abstract Syntax Trees represent the structure of a derivation without the specific details
- Think: "Parse trees without the syntactic sugar"
- Example:

Parse Tree:

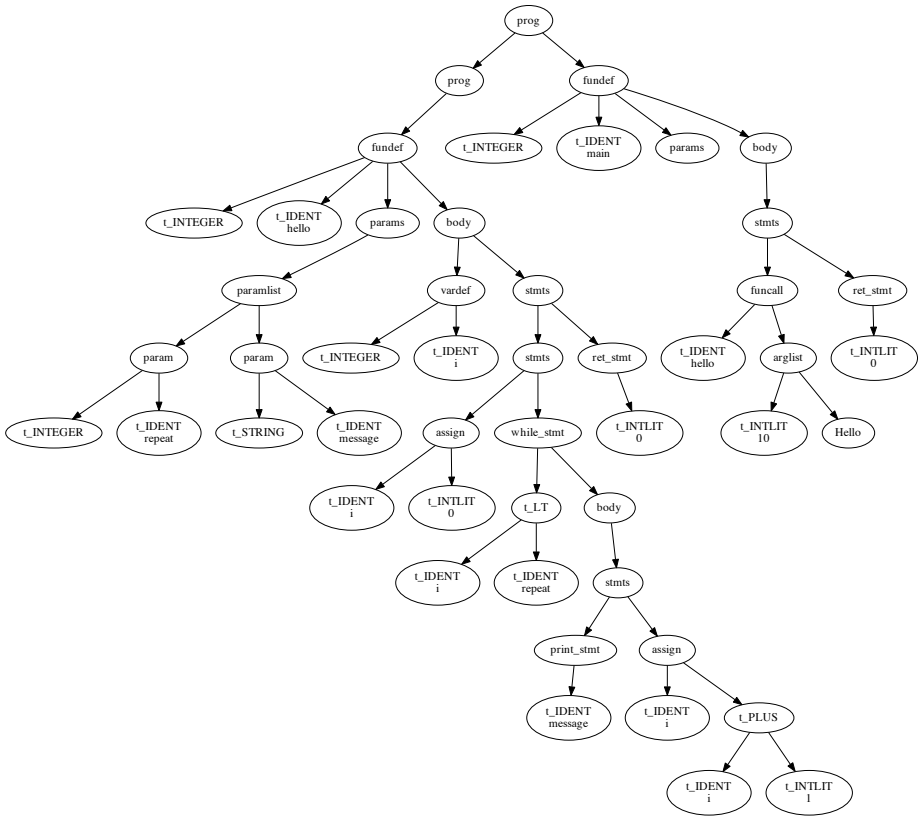



Corresponding AST:

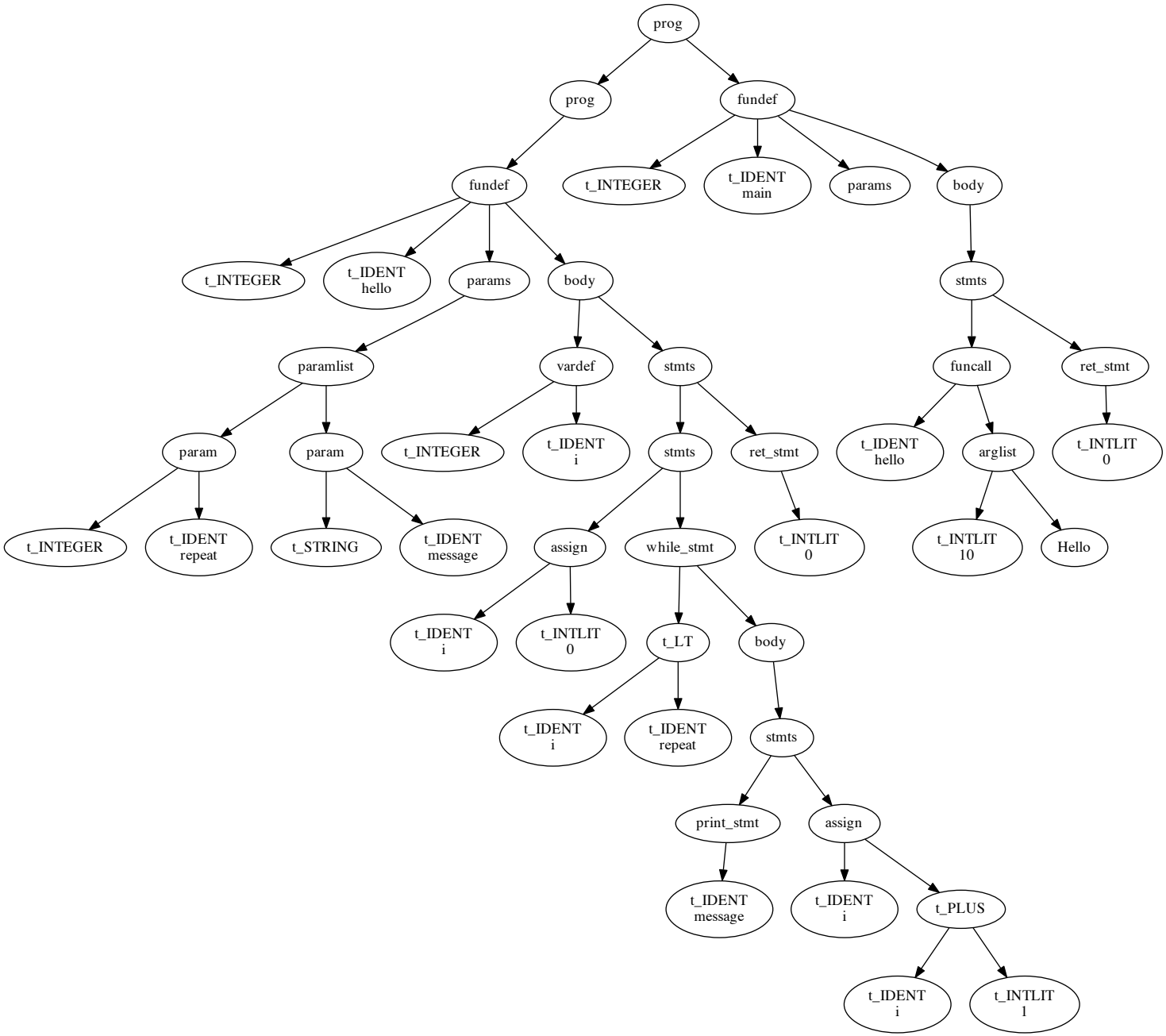


# From Code to AST

```
Integer hello(Integer repeat,
              String message)
{
    Integer i;
    i = 0;
    while(i<repeat)
    {
        print message;
        i = i+1;
    }
    return 0;
}

Integer main()
{
    hello(10, "Hello\n");
    return 0;
}
```



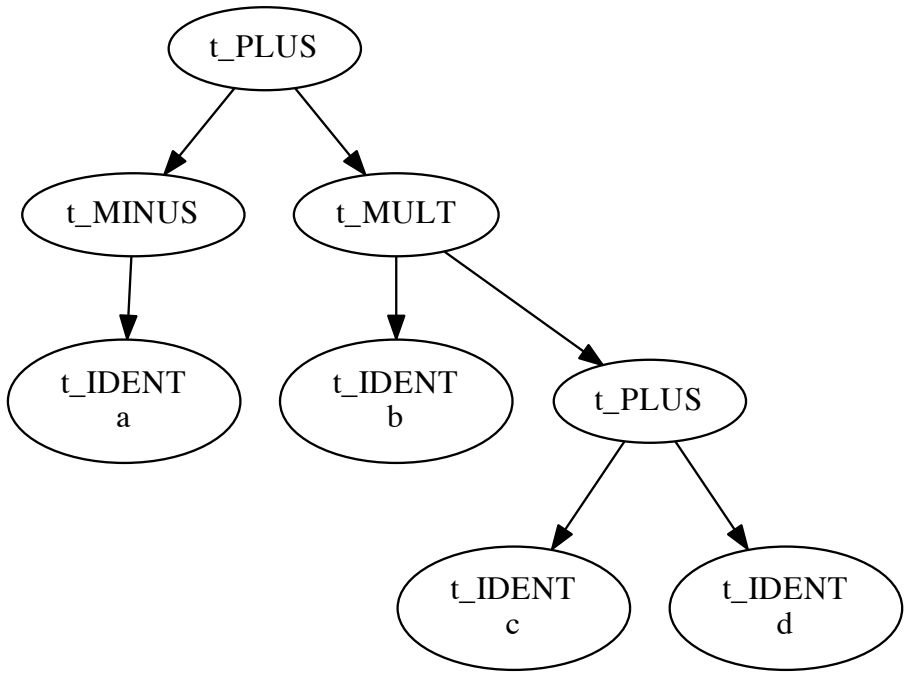
# ....to AST


prog
prog
fundef
fundef
t_INTEGER
t_IDENT
main
params
body
t_INTEGER
t_IDENT
hello
params
body
stmts
paramlist
vardef
stmts
funcall
ret_stmt
param
param
t_INTEGER
t_IDENT
i
stmts
ret_stmt
t_IDENT
hello
arglist
t_INTLIT
0
t_INTEGER
t_IDENT
repeat
t_STRING
t_IDENT
message
assign
while_stmt
t_INTLIT
0
t_INTLIT
10
Hello
t_IDENT
i
t_INTLIT
0
t_LT
body
t_IDENT
i
t_IDENT
repeat
stmts
print_stmt
assign
t_IDENT
message
t_IDENT
i
t_PLUS
t_IDENT
i
t_INTLIT
1

# From Code to AST (smaller example)

Example expression:

$-a+b*(c+d)$




- Observation: Every leaf node corresponds to a lexer token!

# ASTs are Trees

- Trees can be recursively defined:
  - (The empty tree is a tree)
  - A single node is a tree
  - A node whose children are trees is a tree
- For ASTs:
  - Base case: Leaves of the AST are provided by the lexer
  - Recursive case: Internal nodes are build from the subtrees corresponding to the RHS of a matched syntax rule
- In practice:
  - For tokens that correspond to AST leaves: Lexer allocates single node and assigns it to `yylval`
  - For complex rules:
    - Subtrees are ASTs communicated via semantic values of matched RHS terminals or non-terminals
    - Result tree is assembled and passed up via the semantic value of the rule

# Reminder: Desk calculator

- Desk calculator
  - Reads algebraic expressions and assignments
  - Prints result of expressions
  - Can store values in registers `R0-R99`
- Example session:

```
[Shell] ./scicalc
R10=3*(5+4)
> RegVal: 27.000000
(3.1415*R10+3)
> 87.820500
R9=(3.1415*R10+3)
> RegVal: 87.820500
R9+R10
> 114.820500
...
```

# Abstract grammar for desk calculator (partial)

$G_{DC} = \langle V_N, V_T, P, S \rangle$

- $V_T = \{$`PLUS`, `MULT`, `ASSIGN`, `OPENPAR`, `CLOSEPAR`, `REGISTER`, `FLOAT`, $\ldots\}$
  - Some terminals are single characters (+, =, $\ldots$)
  - Others are complex: `R10`, `1.3e7`
  - Terminals ("tokens") are generated by the lexer
- $V_N = \{$`stmt`, `assign`, `expr`, `term`, `factor`, $\ldots\}$

- $P$ :

```
stmt    →  assign
        |  expr
assign  →  REGISTER ASSIGN expr
expr    →  expr PLUS term
        |  term
term    →  term MULT factor
        |  factor
factor  →  REGISTER
        |  FLOAT
        |  OPENPAR expr CLOSEPAR
```

- $S$ = *handwave*
  - For a single statement, $S =$ `stmt`
  - In practice, we need to handle sequences of statements and empty input lines (not reflected in the grammar)

# Bottom-up parsing

- Tokenized input is read left-to-right
- Current state of the parse is represented by a stack
  - Stack contains terminals and non-terminals
- Important operations:
  - *Reduce* - when the top symbols of the stack match the RHS of a rule $LHS \rightarrow RHS$, they can be replaced by the LHS
  - *Shift* - move the next input token from the input onto the stack
  - *Look-ahead* - when both Shift and Reduce are possible, check if a shift would potentially enable a longer RHS to match

# Parsing statements (1)

- Example string: R10 = (4.5+3*7)
- Tokenized: `REGISTER ASSIGN OPENPAR FLOAT PLUS FLOAT MULT FLOAT CLOSEPAR`
  - Abbreviated: R A O F P F M F C
- Parsing state:
  - Unread input (left column)
  - Current stack (top element on right)
  - How state was reached

```
stmt    →  assign
        |  expr
assign  →  R A expr
expr    →  expr P term
        |  term
term    →  term M factor
        |  factor
factor  →  R
        |  F
        |  O expr C
```

| Input | Stack | Comment |
|---|---|---|
| R A O F P F M F C | | Start |
| A O F P F M F C | R | Shift R to stack |
| O F P F M F C | R A | LA: Shift A to stack |
| F P F M F C | R A O | Shift O to stack |
| P F M F C | R A O F | Shift F to stack |
| P F M F C | R A O factor | Reduce F |
| P F M F C | R A O term | Reduce factor |

...

# Parsing statements (2)

| | | |
|---|---|---|
| R A O F P F M F C | | Start |
| A O F P F M F C | R | Shift R to stack |
| O F P F M F C | R A | LA: Shift A to stack |
| F P F M F C | R A O | Shift O to stack |
| P F M F C | R A O F | Shift F to stack |
| P F M F C | R A O factor | Reduce F |
| P F M F C | R A O term | Reduce factor |
| P F M F C | R A O expr | LA: Reduce term |
| F M F C | R A O expr P | Shift P |
| M F C | R A O expr P F | Shift F |
| M F C | R A O expr P factor | Reduce F |
| M F C | R A O expr P term | Reduce factor |
| F C | R A O expr P term M | LA! Shift M |
| C | R A O expr P term M F | Shift F |
| C | R A O expr P term M factor | Reduce F |
| C | R A O expr P term | Reduce tMf |
| C | R A O expr | Reduce ePt |
| | R A O expr C | Shift C |
| | R A factor | Reduce OeC |
| | R A term | Reduce factor |
| | R A expr | Reduce term |
| | stmt | Reduce RAe |

# Reminder: Actions in Bison

- Bison is based on syntax rules with associated actions
  - Whenever a reduce is performed, the action associated with the rule is executed
- Actions can be arbitrary C code
- Frequent: semantic actions
  - The action sets a semantic value based on the semantic value of the symbols reduced by the rule
  - For terminal symbols: Semantic value is `yylval` from Flex
  - Semantic actions have "historically valuable" syntax
    - Value of reduced symbol: `$$`
    - Value of first symbol in syntax rule body: `$1`
    - Value of second symbol in syntax rule body: `$2`
    - ...
    - Access to named components in a union: `$<val>1`

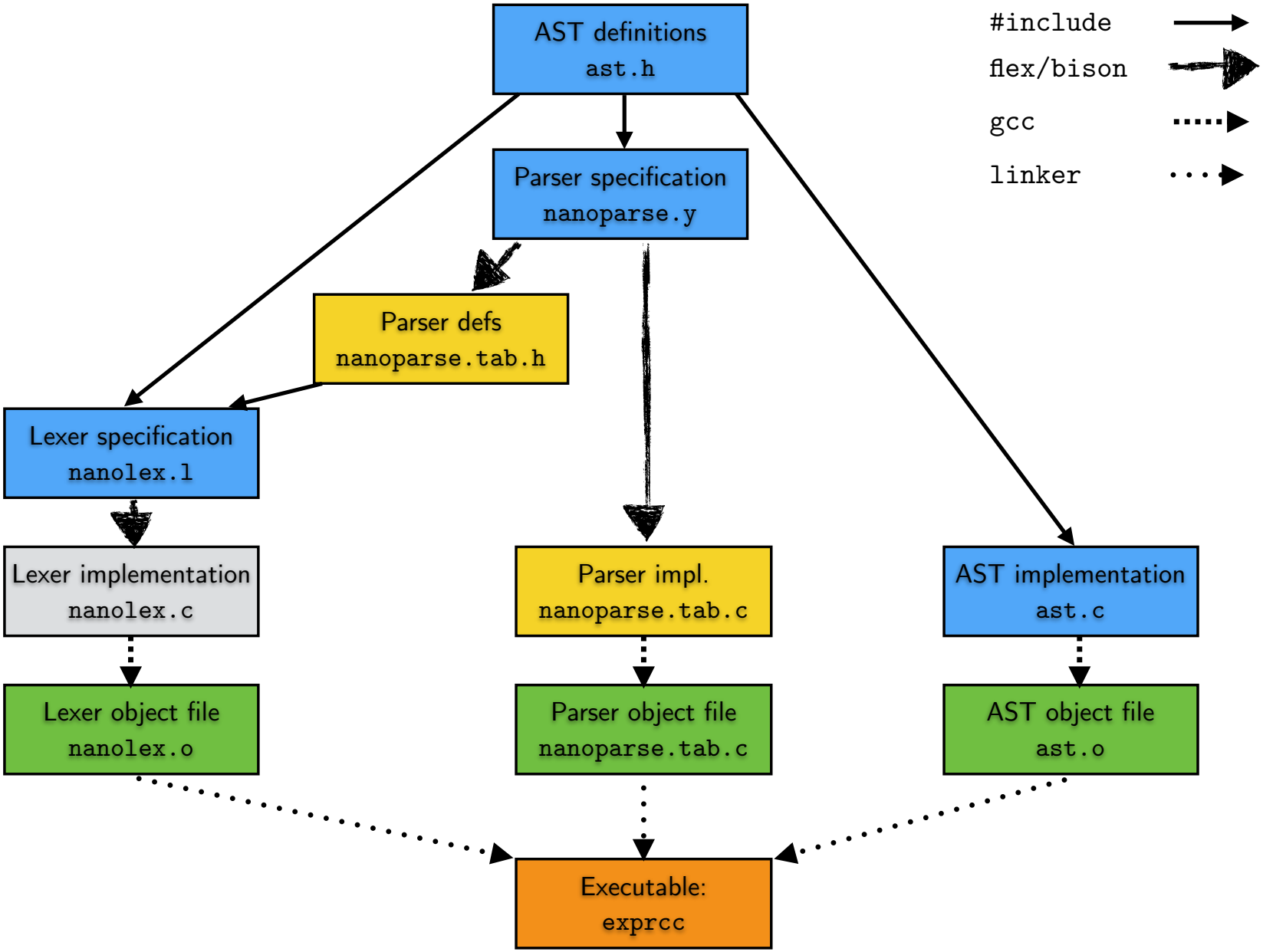
# From text to AST in practice: Parsing *nanoLang* expressions

- Example for syntax analysis and building abstract syntax trees
- Language: *nanoLang* expressions (without function calls)
- Structure of the project
- Building
- Code walk-through

# Exercise: Building exprcc

- Go to
  `http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2018.html`
- Download `NANOEXPR.tgz`
- Unpack, build and test the code
- To test:
  - `./exprcc expr1.nano`
  - `./exprcc --sexpr expr1.nano`
  - `./exprcc --dot expr1.nano`

# exprcc Overview



# `exprcc` Makefile

# Simplified *nanoLang* expression syntax

```
expr: INTLIT
    | IDENT
    | STRINGLIT
    | OPENPAR expr CLOSEPAR
    | expr PLUS expr
    | expr MINUS expr
    | expr MULT expr
    | expr DIV expr
    | MINUS expr
;
```

# Alternative notation

```
expr -> INTLIT
      | IDENT
      | STRINGLIT
      | ( expr )
      | expr + expr
      | expr - expr
      | expr * expr
      | expr / expr
      | - expr
```

Question: Is the grammar unambiguous?

- How do we solve this?

# Precedences and Associativity in Bison

- Code: `nanoparse.y` token definitions
- The trick with unary -

# Implementing ASTs

- Abstract Syntax Trees are trees!
  - Dynamic data structures
  - C: Nodes (structs) and pointers (to children)
  - Dynamic memory management with malloc()/free()
- (Pointers to) AST nodes are used in two contexts
  - Lexer generates AST leaves corresponding to terminal symbols
  - Parser builds complex ASTs from more basic ones
    $\Longrightarrow$ AST datatype needed in Flex and Bison files!
- AST nodes:
  - Type (token or non-terminal represented by node)
  - Unique id (counter)
  - Lexeme (for token AST cells)
  - Numerical value (for integer tokens)
  - Children (array of pointers to AST nodes)
- Code: `ast.h`, `ast.c`

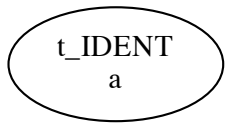
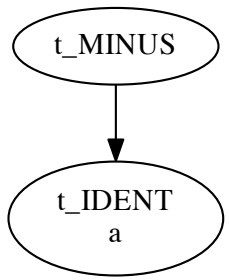
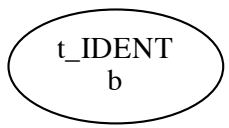
# Lexical Analysis

- Code: `nanolex.l`

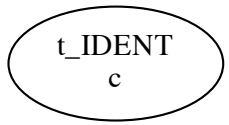
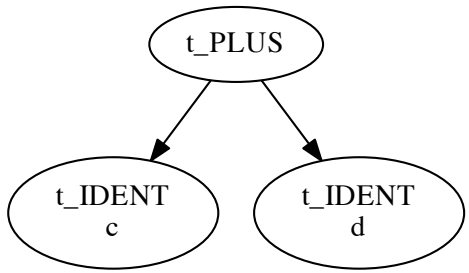
# Building ASTs

- Code: `nanoparse.y` syntax rules and semantic actions

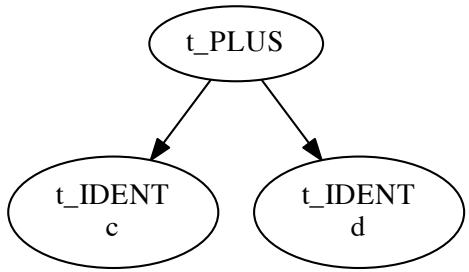
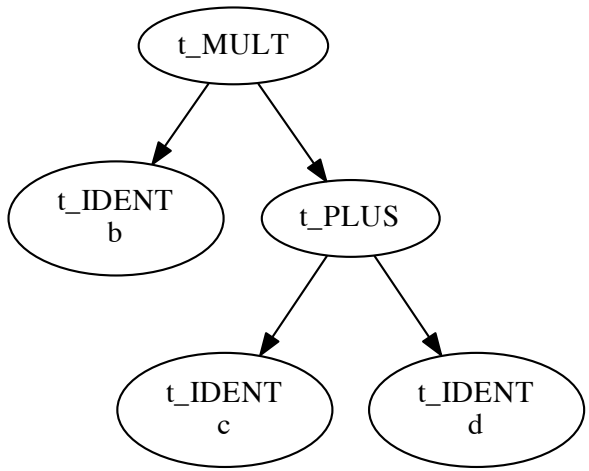
# Parsing in action (1)

| **Unproc.** | **Stack** | **New AST** |
|---|---|---|
| `-a+b*(c+d)` | | (Start of parse) |
| `+b*(c+d)` | `- a` |   |
| Shift, Shift, Reduce `IDENT` to `expr` | | |
| `+b*(c+d)` | `- expr` |   |
| Reduce `-expr` to `expr` | | |
| `*(c+d)` | `expr + b` |   |
| Shift, Shift, Reduce `IDENT` to `expr` | | |

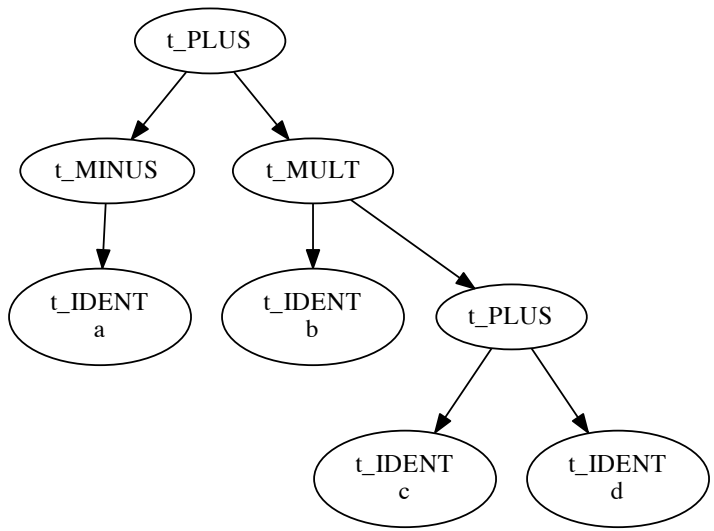
# Parsing in action (2)

| Unproc. | Stack | New AST |
|---|---|---|
| `+d)` | `expr + expr * ( c` |  |
| Shift, Shift, Shift, Reduce `IDENT` to `expr` | | |
| `)` | `expr + expr * ( expr + d` |  |
| Shift, Shift, Reduce `IDENT` to `expr` | | |
| `)` | `expr + expr * ( expr + expr` |  |
| Reduce `expr+expr` to `expr` | | |

# Parsing in action (3)

| Unproc. | Stack | New AST |
|---|---|---|
| Shift, Reduce `(expr)` to `expr` | | |
| | `expr + expr * ( expr )` |  |
| Reduce `expr*expr` to `expr` | | |
| | `expr + expr * expr` |  |

# Parsing in action (4)

| Unproc. | Stack | New AST |
|---|---|---|
| | `expr + expr` |  |



Reduce `expr+expr` to `expr`

# Details: Error Handling and Locations

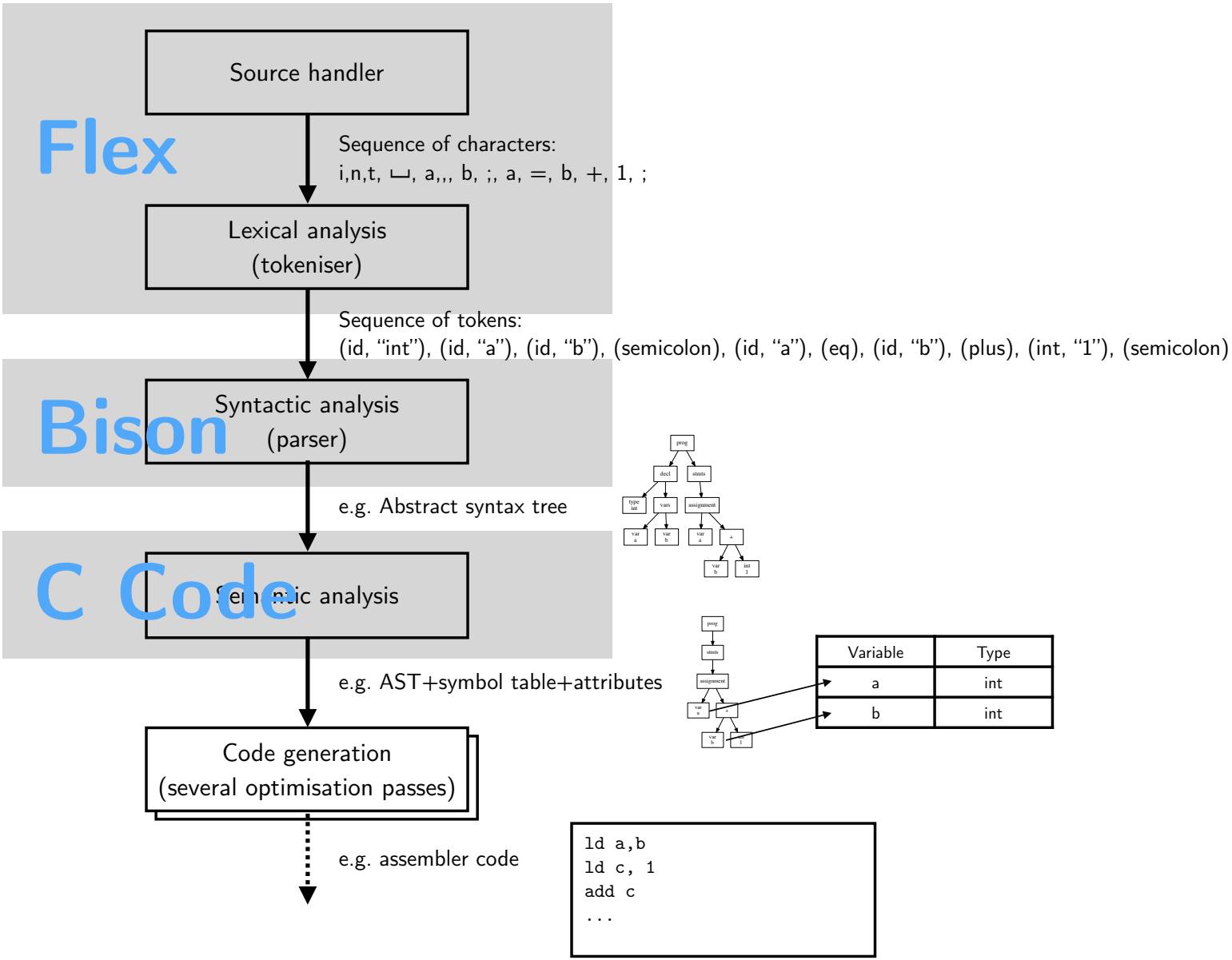
- Bison supports locations with `%locations`
  - Global variable `yylloc`

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

  - Automagically tracked by parser, to be set by lexer for tokens!
  - Can be used in `yyerror()`
- Bison parsers call `yyerror(const char* err)` on error
  - Argument is provided by parser
  - Location of error can be read out of `yylloc`
  - `%define parse.error verbose` usually improves error reporting
- Code: `nanolex.l` position tracking, `nanoparse.y` error handling

End Lecture 4

# High-Level Architecture of a Compiler



# Semantic Constraints

# Group Exercise: Spot the Bugs (1)

```
Integer fun1(Integer i, Integer i)
{
    Integer i;

    if(i > 0)
    {
        print j;
    }
}

Integer main()
{
    fun1(1, 2);
    fun2(1, 2);
    return 0;
}
```

# Group Exercise: Spot the Bugs (2)

```
Integer fun1(Integer i, Integer j)
{
    Integer i;

    if(i > "0")
    {
        print j+"12";
    }
    return 1;
}

Integer main()
{
    fun1(1, "Hello");
    fun1(1, 2, 3);
    return 0;
}
```

# Group Exercise: Spot the Bugs (3)

```
Integer fun1(Integer i, Integer j)
{
    while(j>i)
    {
        Integer j;

        print j;
        j=j+1;
    }
    return 1;
}
```

# Semantic constraints of *nanoLang* (V 1.0)

- Every name has to be defined before it can be used
- Every name can only be defined once in a given scope
- Functions must be called with arguments of the right type in the right order
- Operands of comparison operators must be of the same type
- Operands of the arithmetic operators must be of type `Integer`
- Every program must have a `main()`
- (Every function must have a `return` of proper type)

# Types in General

The type system is one of the defining elements of a programming language!

- Atomic types: Directly supported by the language
  - Integers
  - Floating point numbers
  - Characters
  - Strings
  - .....
- Derived/composite types
  - Structs/records - combine several other types into one new structure
  - Arrays - combine many elements of the same type into one structure
  - Pointers/References
  - Function types (maps input types to output type)

# Types in *nanoLang*

- *nanoLang* has a very basic type system!
- Two atomic types
  - Integer
  - String
- One way to construct new types
  - Functions

Still an infinite number of types!

- `→Integer`
- `Integer→Integer`
- `Integer, Integer→Integer`
- ...

# Managing Symbols

- Properties of identifiers are stored in a symbol table
  - Name
  - Type
- Properties of identifiers depend on part of the program under consideration!
  - Names are only visible in the scope they are declared in
  - Names can be redefined in new scopes

Symbol tables need to change when traversing the program/AST for checking properties and generating code!

# Names and Variables

## Definition (Variable)

A variable is a location in memory (or "in the store") that can store a value (of a given type).

- Variables can be statically or dynamically allocated
  - Typically: global variables are statically allocated (and in the data segment of the process
  - Local variables are dynamically managed and on the stack
  - Large data structures and objects are stored in the heap

## Definition (Name)

A name is an identifier that identifies a variable (in a given scope).

- The same name can refer to different variables (recursive function calls)
- Different names can refer to the same variables (depends on programming languages - aliasing)

# Scopes and Environments

- The environment establishes a mapping from names to variables
- Static scope: Environment depends on block structure of the language
  - In any situation, the name refers to the variable defined in the nearest surrounding block in the program text
  - Examples: C (mostly), Pascal, Java, modern LISPs (mostly)
- Dynamic scope: Environment depends on calling sequence in program
  - Name refers to the same variable it referred to in the calling function
  - Traditional LISP systems (Emacs LISP)

# Group exercise: Static and dynamic scopes in C

```c
#include <stdio.h>
int a=10;
int b=10;
#define adder(x) (x)+a

void machwas(int a, int c)
{
    printf("adder(a)=%d\n", adder(a));
    printf("adder(b)=%d\n", adder(b));
    printf("adder(c)=%d\n", adder(c));
    {
        int c = 5;
        printf("adder(c)=%d\n", adder(c));
    }
}

int main(void)
{
    machwas(1, 2);
    machwas(2, 3);

    return 0;
}
```

# Example: Scopes in *nanoLang*

```
Integer i;

Integer fun1(Integer loop)
{
    Integer i;

    i=0;
    while(i<loop)
    {
        i=i+1;
        print "Hallo";
    }
    return 0;
}

Integer main()
{
    i = 5;
    fun1(i);
}
```

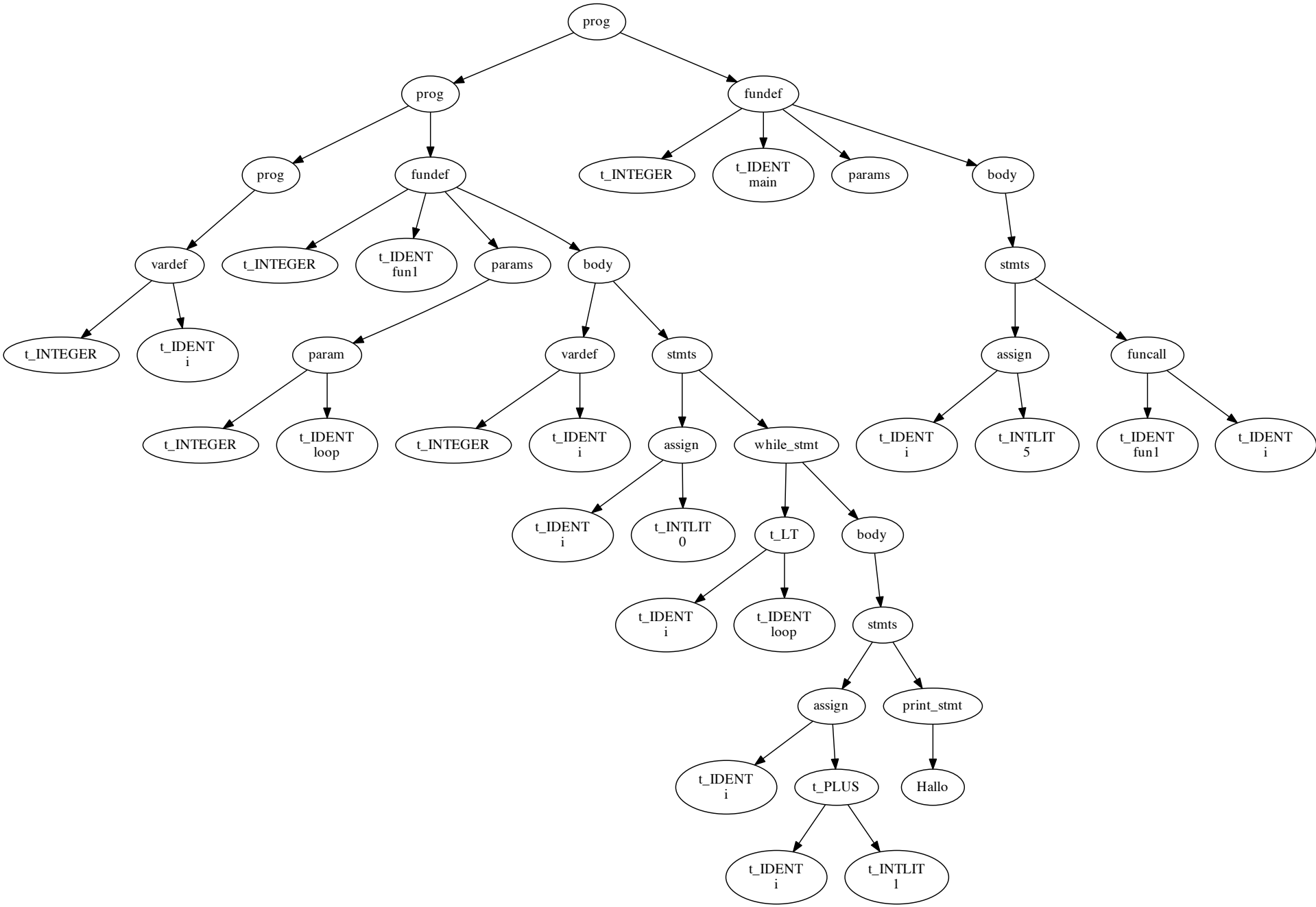
# Scopes in *nanoLang*

- Global scope
  - Global variables
  - Functions
- Function scope
  - Function parameters
- Block scope
  - block-local variables

# Example: Scopes in *nanoLang*

```
Integer i;

Integer fun1(Integer loop)
{
    Integer i;

    i=0;
    while(i<loop)
    {
        i=i+1;
        print "Hallo";
    }
    return 0;
}

Integer main()
{
    i = 5;
    fun1(i);
}
```

# Walking the AST



# Static type checking

- Types are associated with names
- Types are checked at compile time or development time
- Advantages:
  - ?
- Disadvantages:
  - ?
- Typically used in:
  - ?

# Dynamic type checking

- Types are associated with values
- Types are checked at run time
- Advantages:
  - ?
- Disadvantages:
  - ?
- Typically used in:
  - ?

# No type checking

- Programmer is supposed to know what (s)he does
- Types are not checked at all
- Advantages:
  - ?
- Disadvantages:
  - ?
- Typically used in:
  - ?

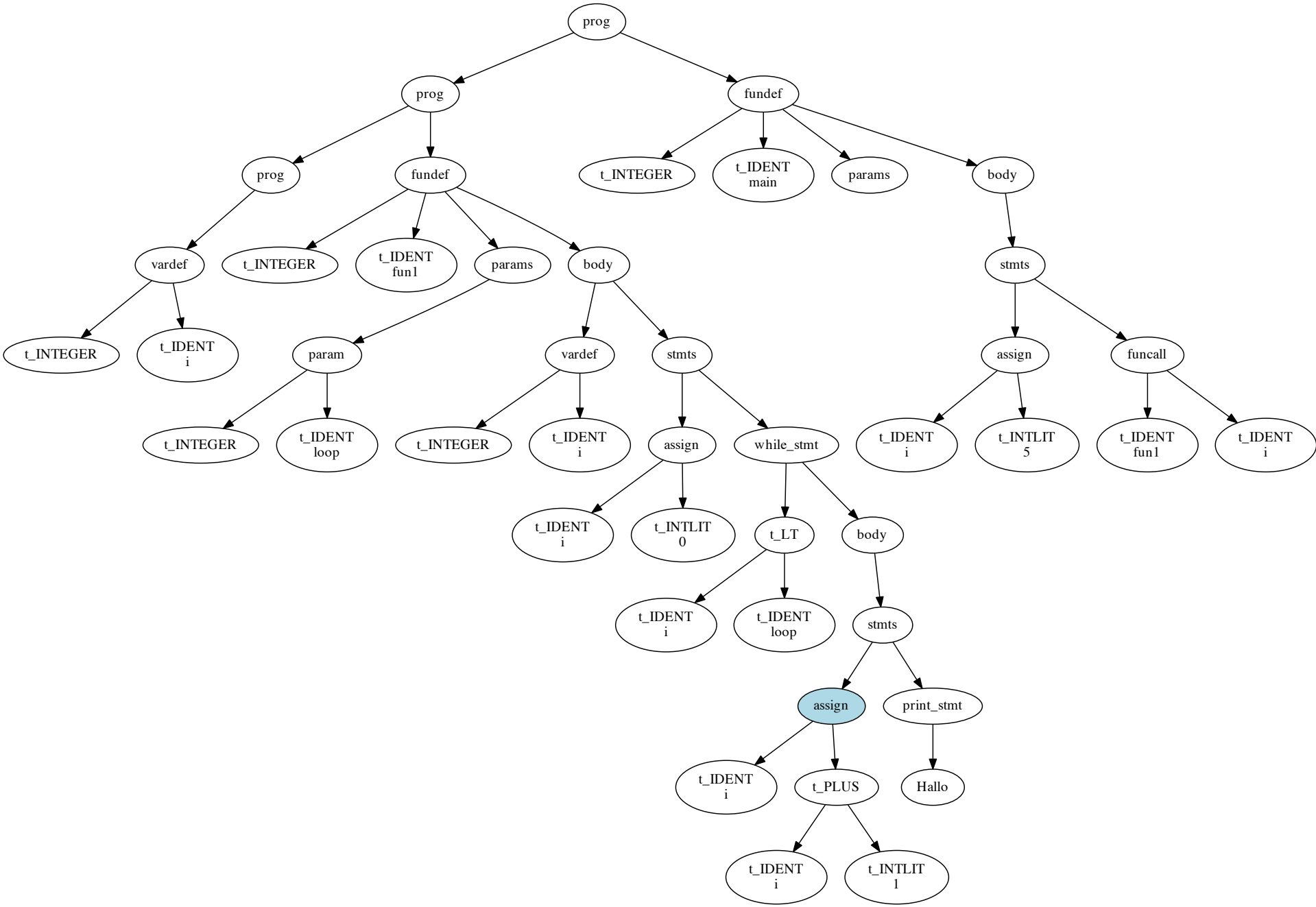
# Exercise: How many types occur in this example?

```
Integer i;

Integer fun1(Integer loop)
{
    Integer i;

    i=0;
    while(i<loop)
    {
        i=i+1;
        print "Hallo";
    }
    return 0;
}

Integer main()
{
    i = 5;
    fun1(i);
}
```

# Symbol tables

- Store name (identifier) and type
- Nested for each scope:
  - Global scope
  - Each new scope entered will result in a new symbol table for that scope, pointing to the preceeding (larger scope)
- Local operations:
  - Add name/type (error if already defined)
- Global operations:
  - Find name (finds "nearest" matching entry, or error)
  - Enter new scope (creates a new empty symbol table pointing to the predecessor (if any)
  - Leave scope (remove current scope)

# Walking the AST



# Representing types

- Types table:
  - Numerical encoding for each type
  - *Recipe* for each type
    - *nanoLang* basic types are atomic
    - Atomic types can also be addressed by name
    - Function types are vectors of existing types
- E.g.

| **Encoding** | **Type** | **Recipe** |
|---|---|---|
| 0 | `String` | atomic |
| 1 | `Integer` | atomic |
| 2 | `Integer fun(Integer, String)` | (1, 1, 0) |

- Operations:
  - Find-or-insert type
    - Return encoding for a new type
    - If type does not exist yet, create it

End Lecture 5

# The Big Picture: Type Checking

- We need to know the type of every expression and variable in the program
  - . . . to detect semantic inconsistencies
  - . . . to generate code
- Some types are simple in *nanoLang*
  - String constants are type `String`
  - Integer constants are type `Integer`
  - Results of arithmetic operations are `Integer`
- Harder: What to do with identifiers?
  - Type of the return value of a function?
  - Types of the arguments of a function?
  - Types of the values of a variable?

The answers depend on the definitions in the program!

# Symbol Tables

- Symbol tables associate identifiers and types
- Symbol tables form a hierarchy
  - Symbols can be redefined in every new context
  - The "innermost" definition is valid
- Symbol tables are filled top-down
  - Outermost symbol-table contains global definitions
  - Each new context adds a new layer
  - Search for a name is from innermost to outermost symbol table

# Building Symbol Tables

Program

```
Integer i;

Integer fun1(Integer loop)
{
   Integer i;

   i=0;
   while(i<loop)
   {
      i=i+1;
      print "Hallo";
   }
}

Integer main()
{
   i = 5;
   fun1(i);
}
```

Symbol table

| | |
|---|---|
| i | Integer |
| fun1 | (Integer)→Integer |
| main | ()→Integer |
| loop | Integer |
| i | Integer |
| - | - |

| | |
|---|---|
| i | Integer |
| fun1 | (Integer)→Integer |
| main | ()→Integer |
| - | - |
| - | - |

# Simplified Type Handling

- Handling complex types directly is cumbersome
- Better: Manage types separately
  - Types are stored in a separate table
  - Symbol table only needs to handle indices into type table

# Symbol Tables and Type Tables

Program

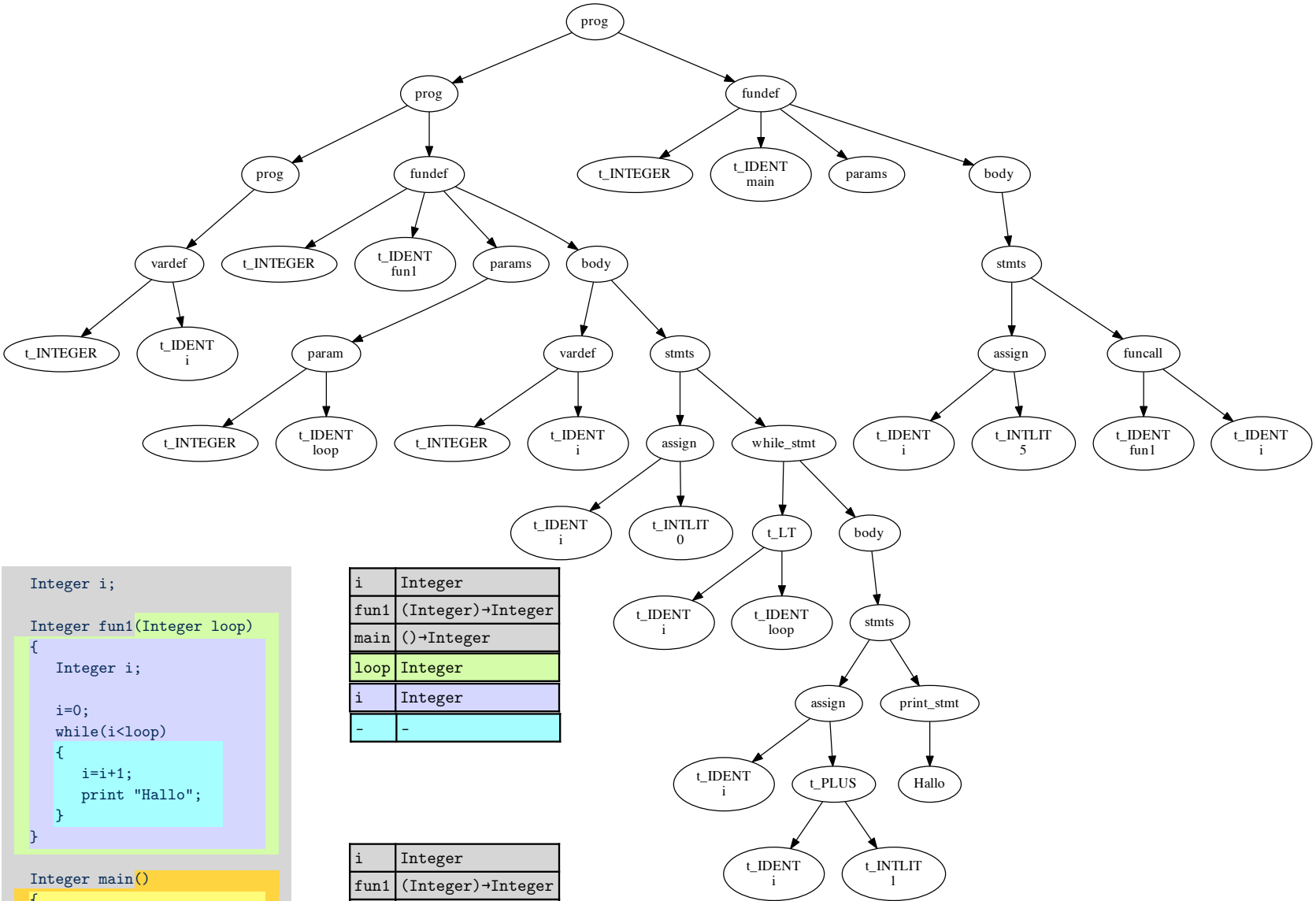
```
Integer i;

Integer fun1(Integer loop)
{
   Integer i;

   i=0;
   while(i<loop)
   {
      i=i+1;
      print "Hallo";
   }
}

Integer main()
{
   i = 5;
   fun1(i);
}
```

Symbol table

| | |
|---|---|
| i | Integer |
| fun1 | (Integer)→Integer |
| main | ()→Integer |
| loop | Integer |
| i | Integer |
| - | - |

| | |
|---|---|
| i | Integer |
| fun1 | (Integer)→Integer |
| main | ()→Integer |
| - | - |
| - | - |

# Symbol Tables and the AST

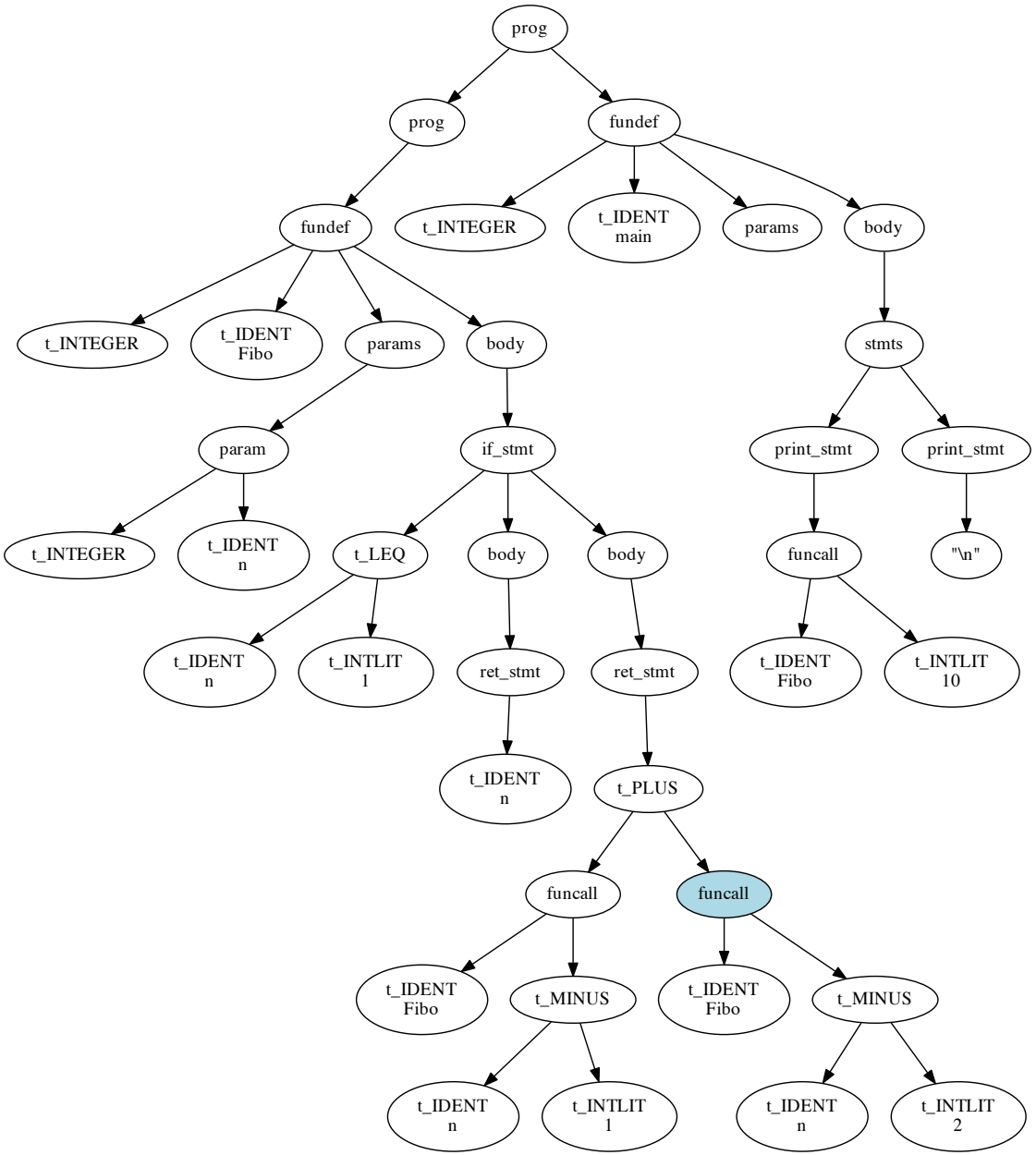



```
Integer i;

Integer fun1(Integer loop)
{
   Integer i;

   i=0;
   while(i<loop)
   {
      i=i+1;
      print "Hallo";
   }
}

Integer main()
{
   i = 5;
   fun1(i);
}
```

| i | Integer |
|---|---|
| fun1 | (Integer)→Integer |
| main | ()→Integer |
| loop | Integer |
| i | Integer |
| - | - |

| i | Integer |
|---|---|
| fun1 | (Integer)→Integer |
| main | ()→Integer |
| - | - |
| - | - |

# Exercise: Symbol tables and the AST




1. Reconstruct the *nanoLang* code corresponding to the AST
2. Identify all different scopes in the program
3. Identify all different scopes in the AST
4. Generate and fill the hierarchy of symbol tables valid at the blue node

# Type Inference in *nanoLang*

- Goal: Determine the (result) type of every expression in the program
- Process: Process AST `bottom-up`
  - Constants: "natural" type
  - Variables: Look up in symbol table
  - Function calls: Look up in symbol table
    - If arguments are not of proper type, error
    - Otherwise: return type of the function
  - Arithmetic expressions:
    - If arguments are `Integer`, result type is `Integer`
    - Otherwise: error

# Contrast: Aspects of Type Inference in C

- Arithmetic expressions:
  - Roughly: arithmetic types are ordered by size (`char` $<$ `int` $<$ `long` $<$ `long long` $<$ `float` $<$ `double`)
  - Type of `a + b` is the greater of the types of `a` and `b`
- Arrays
  - If `a` is an array of `int`, than `a[1]` is of type `int`
- Pointers
  - If `a` is of type `char*`, then `*a` is of type `char`
- Many more cases:
  - Structures
  - Enumerations
  - Function pointers

# Implementation Examples

- `main()` in `nanoparse.y`
- `STBuildAllTables()` in `semantic.c`
- `symbols.h` and `symbols.c`
- `types.h` and `types.c`

End Lecture 6

# Typechecking the AST (1)

- Reminder: We have several goals
  - Assign a type to every AST node which is a subexpression
    - Needed for type checking
    - Needed for code generation
  - Typecheck all expressions
    - Catch and flag type errors
  - (Check proper use of return values)
    - Catch and flag type and/or logic errors

Practical algorithm?

# Typechecking the AST (2)

- Observation: Type of a *nanoLang* AST node can be determined without knowing the type of the subtrees below it
  - Result type of a `t_PLUS/+` node is always `Integer`
  - Result type of a function is known from its type declaration
  - ...
- This is not in general true for other languages!
  - Example: Type of +-expression in C depends on type of arguments
  - Example: Operator overloading in C++
  - General problem: Polymorphism
- Solution: Perform type checking and type assignment at the same time

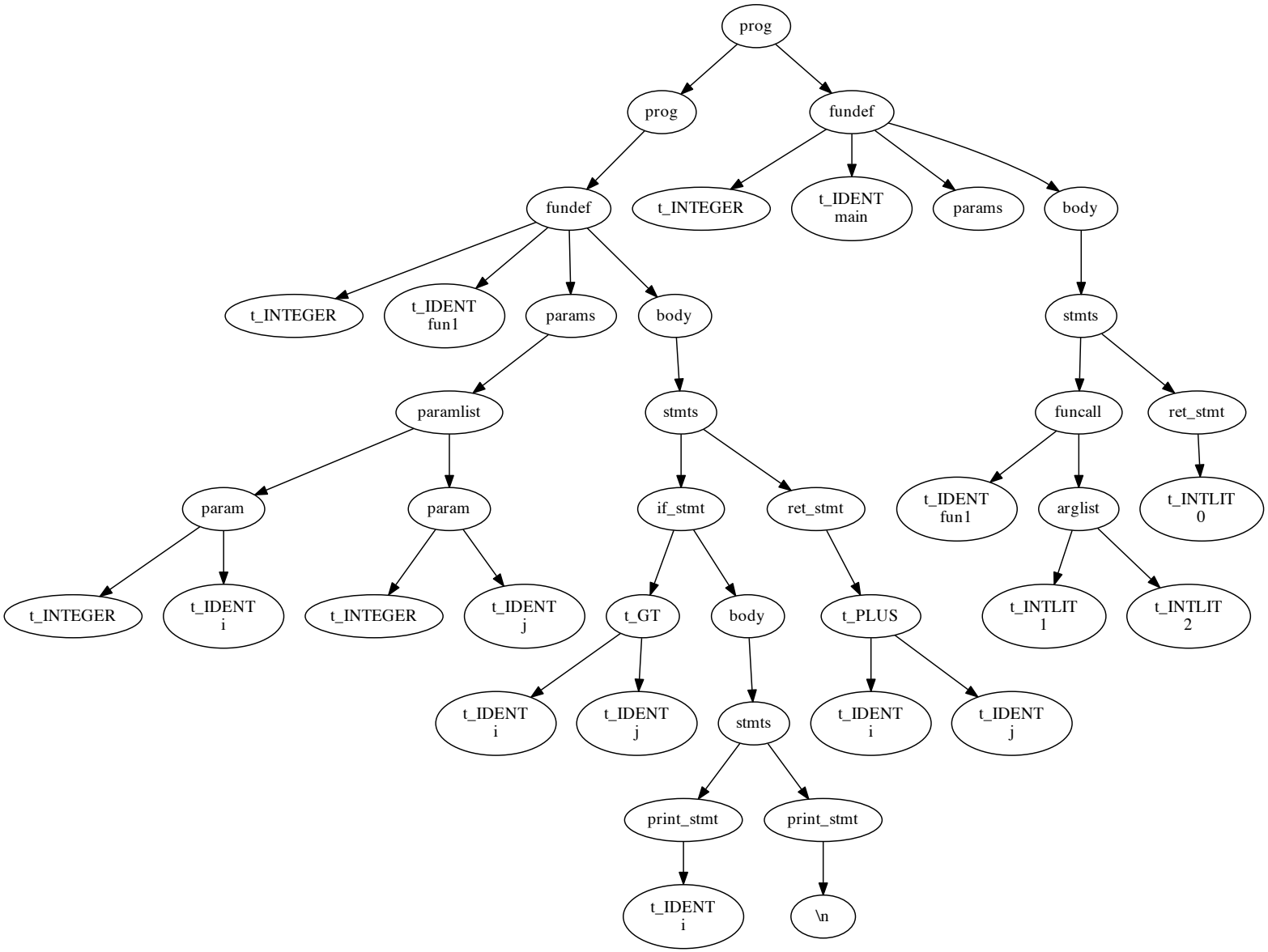
# Typechecking the AST (3)

- Type assignment and type checking for *nanoLang*
  - Input: `ast` is root of an AST
  - Assumption: AST is annotated with symbol tables
- Recursive algorithm
  1. Type check all children of `ast` and assign them their types
     - Note: There may be zero children - this is where the recursion stops!
  2. Perform case distinction on the current AST node `ast`
     - Arithmetic operator: Check if both arguments are `Integer`, result type is `Integer`
     - Comparison operator: Check if both arguments have the same type, result type is `NoType`
     - String literal: Result type is `String`
     - Integer literal: Result type is `Integer`
     - Assignment: Check if type of the identifier on the left is the same as the type of the expression on the right. Result type is `NoType`
     - Function call: Check that argument types are correct (using symbol table), return type according to symbol table
     - ...
     - Body, Prog, vardef...: Result type is `NoType`

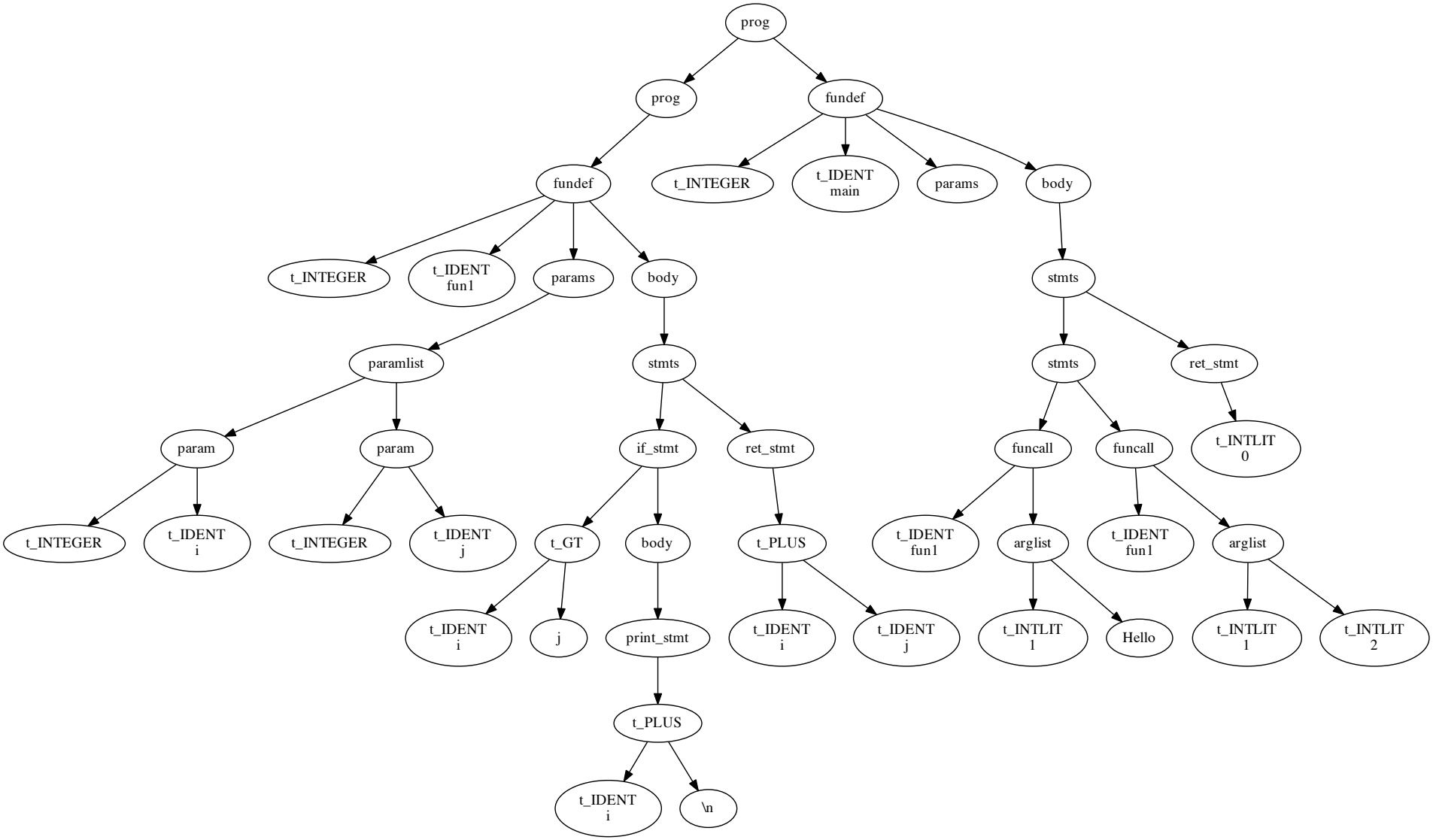
# Exercise: Typing and Typechecking (1)

- For the following two ASTs, use the approach described above to ...
  - ... determine the result type for each AST node
  - ... find and mark all type errors (if any)
- Bonus: Think of an approach to check correct use and typing of `return`

# Exercise: Typing and Typechecking (2)



# Exercise: Typing and Typechecking (3)



# Excursion: `assert()`

- `assert()` is a facility to help debug programs
  - Part of the C Standard since C89
  - To use, `#include <assert.h>`
- `assert(expr)` evaluates `expr`
  - If `expr` is false, then an error message is written and the program is aborted
  - Otherwise, nothing is done
- Hints:
  - Particularly useful to check function parameter values
  - To disable at compile time, define the macro `NDEBUG` (e.g. with the compiler option `-DNDEBUG`
  - Useful idiom: `assert(expr && "What's wrong");`
  - More information: `man assert`

# Semantics of Compiled and Target Language

- Before we can compile a language, we must understand its semantics
- Important questions:
  - How are parameter passed into functions?
  - Related: How do assignments work?
- Before we can compile a language, we must understand the target language and environment
  - How are parameters and local variables handled?
  - How does the program interact with the OS and the environment?

# Parameter Passing

- Call by value
  - Formal parameters become new local variables
  - Actual parameters are evaluated and used to initialize those variables
  - Changes to variables are irrelevant after function terminates
- Call by reference
  - Only *references* to existing variables are passed
  - In effect, formal parameters are bound to *existing* variables
  - Actual parameters that are not variables themselves are evaluated and placed in anonymous new variables
  - Changes to parameters in functions change the original variable
- ~~Call by name~~
  - Only historically interesting
  - Semantics mostly similar to *call-by-value*

# Parameter Passing - Advantages and Disadvantages?

- Call by value?
- Call by reference?
- For your consideration:

```
int fun(int a, int b)
{
    a++;
    b++;
    return a+b;
}

int main(void)
{
    int i=0;

    fun(i,i);
    printf("i=%d\n", i);
}
```

# Parameter Passing in C/C++/Pascal/Scheme?

- C?
- C++?
- Pascal?
- LISP/Scheme?
- Others?

# Assignments

- What happens if `a = b;` is encountered?
  - If both are integer variables (or values)?
  - If both are string variables (or values)?
  - If both have an object type?
  - If both are arrays?

# Calling Conventions

- How are parameters values passed at a low level?
  - Registers?
  - Stack?
  - Other?
- Who is responsible for preserving registers?
  - Caller?
  - Callee?
- In which order are parameters passed?
- How is the old context (stack frame and program counter) preserved and restored?

For our *nanoLang* compiler, we rely on C to handle these things!

# Runtime System and OS Integration

- Runtime system provides the glue between OS and program
  - Translates OS semantics/conventions to compiled language and back
- Runtime system provides execution support for program semantics
  - Higher-level functions/data types
  - Memory management
  - Library functions

# Parameter Passing and Assignments in *nanoLang*

- Suggestion: All parameter passed "as if" by value
- Integer: Pass by value
- *Immutable strings*
  - Can be passed by reference
  - Need to be memory-managed (reference counting, a job for the runtime system)
  - Alternative is not simpler!

# nanoLang OS Integration

- Command line arguments
  - Suggestion: `main()` takes arbitrary number of string arguments
  - These are filled from the command line
  - Spare arguments are represented by the empty string
- Exit and return value
  - Library function `Exit(val)` terminates program and returns integer value
  - `return` from `main()` has the same effect

# *nanoLang* Library Functions

- Suggested function to make things interesting:
  - `StrIsInt(str)`: Returns 1 if `str` encodes a valid integer, 0 otherwise
  - `StrToInt()`: Converts a string to an integer. If `str` is not an integer encoding, result is undefined
  - `IntToStr(int)`: Returns a string encoding of the given integer
  - `StrLength(str)`: Returns the lengths of str
- More suggestions?
  - `String StrFront(str, int)` - return first `int` characters as new string
  - `String StrRest(str, int)` - return all but first `int` characters
  - `String StrCat(str, str)` - concatenate strings, return as new string
  - `Integer StrToASCII(str)` - only for strings of lenght 1, return ASCII value
  - `String ASCIIToStr(int)` - reverse of the above

# *nanoLang* Strings

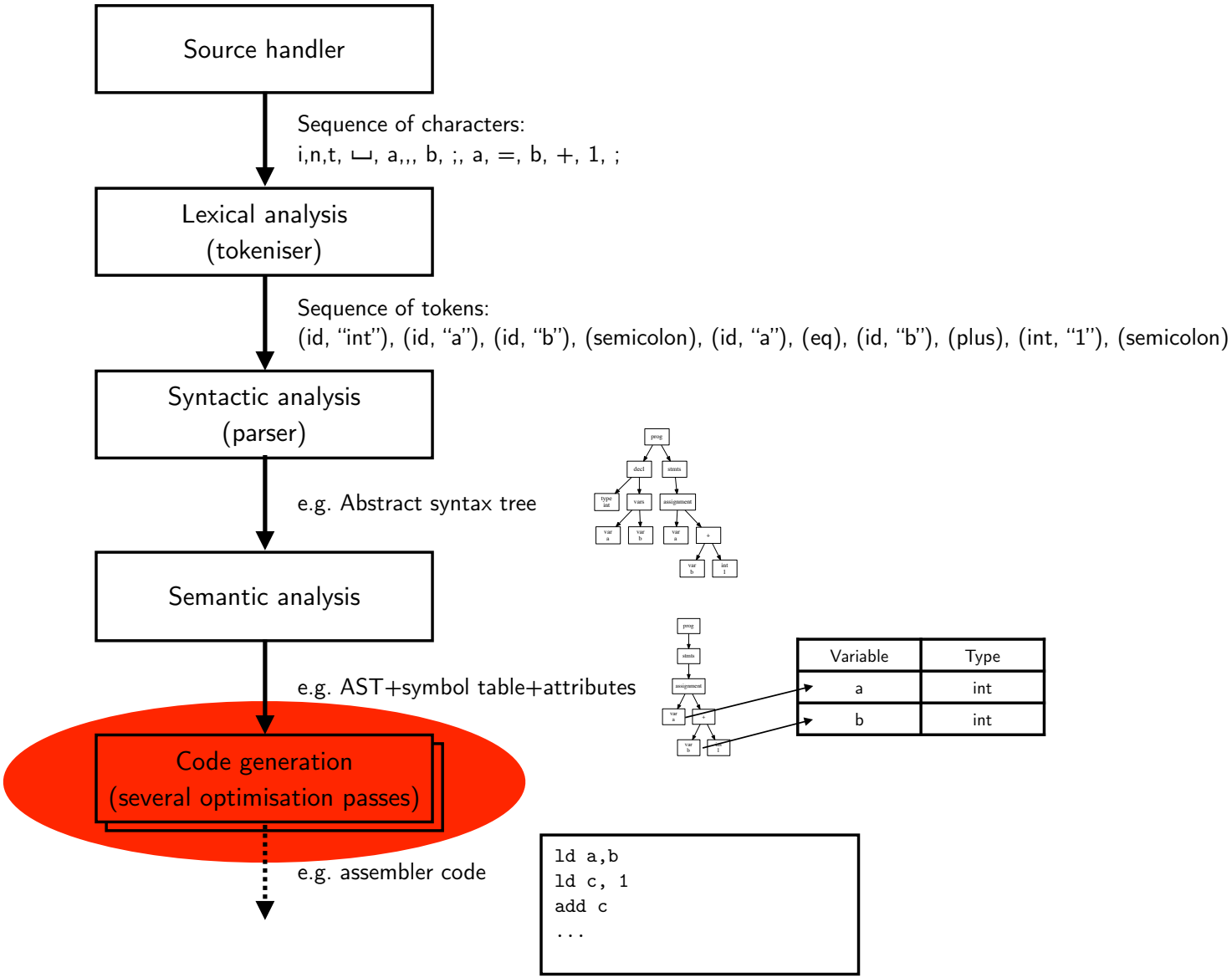
- Temptation: Use C `char*`
- Fails as soon as strings can be dynamically created
- Suggestion: Structure with reference counting
  - String value - the actual string (`malloc()`ed `char*`)
  - Length (maybe)
  - Reference count - how many places have a reference to the string?
    - Increase if string is assigned to a variable or passed to a function
    - Decrease, if a variable is reassigned or goes out of scope
    - Free string, if this reaches 0

End Lecture 7

# Coding Hints

- The *nanoLang* compiler is a non-trivial piece of software
  - Several modules
  - Several different data types (AST, Types, Symbols)
- It helps to follow good coding practices
  - The big stuff: Good code structure
    - One function per function
    - Not more than one screen page per function
  - The small stuff
    - Clean formatting (including vertical space)
    - Use expressive names for functions and variables
    - Reasonable comments (don't over-comment, though!)
    - Use `assert()`
    - Compile with warnings enables (Makefile: `CFLAGS = -Wall`)

# The Final Phase



# Code Generation `nanoLang` to C

- Suggestion: Code generation uses separate phases
  - Initial boilerplate
  - Global variable definitions
  - Function declarations
  - Constant library code
  - Translation of function definitions
  - C `main()` function

# Name Mangling

- To avoid name conflicts, *nanoLang* identifers should use a standard naming scheme that guarantees that they don't conflict with internal C names
- Suggestion:
  - Atomic type names are prepended with `N_`
  - Function and variable names are prepended with `n_`

Compare C (from ISO/IEC 9899:1999/C99):

- All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.
- All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

# Initial Boilerplate

- Emit constant code needed for each translated *nanoLang* program
  - Comment header
  - Standard system includes
  - Type definitions
  - Possibly macro definitions
- Implementation via printing constant string
  - Easiest way
  - Alternative: Read from file

# Global Variable Definitions

- Visibility difference between *nanoLang* and C
  - Globally defined *nanoLang* identifiers are visible throughout the program
  - C definitions are visible from the point of definition only
  - Hence we need to declare variables (and functions) upfront
- Implementation suggestion:
  - Iterate over all symbols in the global symbol table
  - For each variable symbol, emit a declaration

# Function Declarations

- The same visibility difference between *nanoLang* and C affects functions
  - We need to declare all functions upfront!
- Implementation suggestion:
  - Iterate over all symbols in the global symbol table
  - For each function symbol, emit a declaration

Suggestion: For simplicity and consistency, we should insert the *nanoLang* standard library functions (`Exit()`, `StrIsInt()`, `StrToInt`, ...) into the symbol table (and do so before semantic analysis to stop the user from inadvertently redefining them!)

# Constant Library Code

- The *nanoLang* runtime will need various pieces of code
  - Data types and helper functions to handle e.g. Strings
  - Implementations of the build-in functions
- Implementation options
  - Just insert plain C code here (Alternative 0, but this may be lengthy)
  - Alternative 1: Read this C code from a file
  - Alternative 2: Just `#include` the full C code
  - Alternative 3: `#include` only header with declarations, then require linking with a run time library later

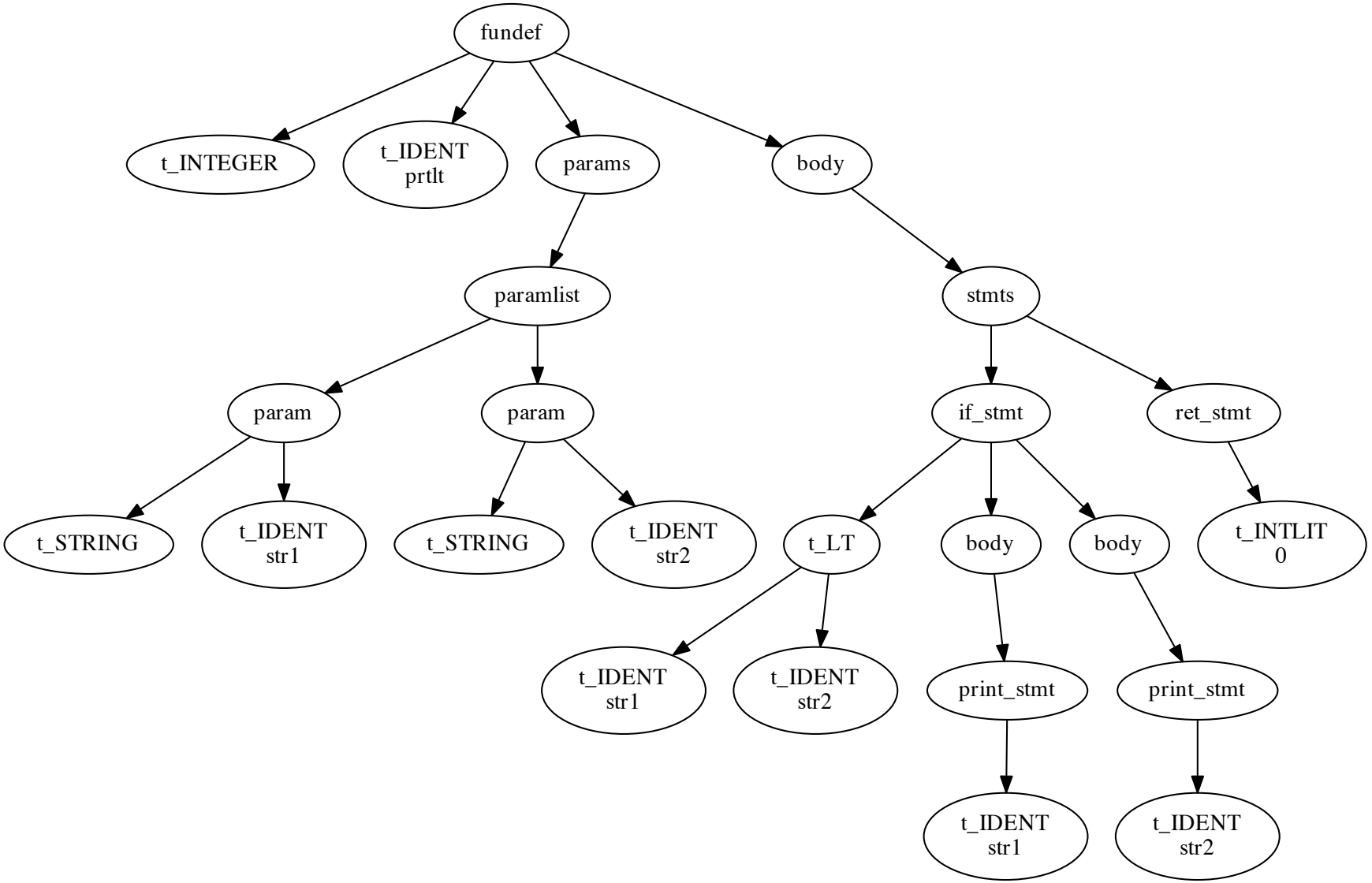
# Translation of Function Definitions

- This is the heart of the compiler!
- Go over the AST and emit a definition for each function
  - *nanoLang* functions become C functions
  - Local *nanoLang* variables become C variables of an appropriate type
  - *nanoLang* blocks become C blocks
  - *nanoLang* instructions are translated into equivalent C statement sequences
  - Mostly straightforward
    - `print` requires case distinction
    - String comparisions require library calls

More complex: Proper string handling

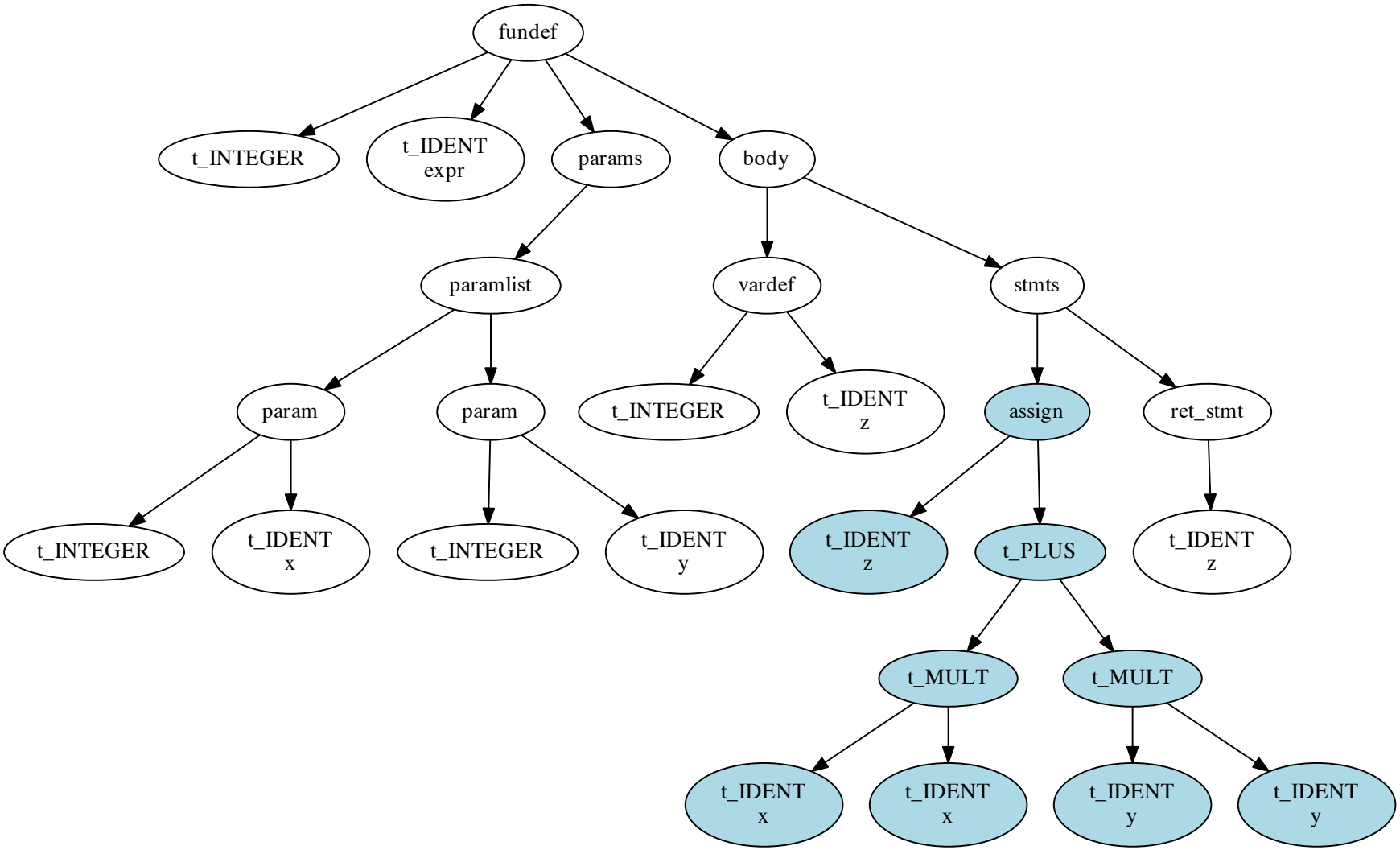
# Exercise: Code Generation (1)

Generate C code for the following AST:



# Exercise: Code Generation (2)

Generate C code for the coloured part of the following AST:



# C `main()` Function

- Generate an appropriate `main()` function
- Tasks:
  - Read commandline and initialize parameters for *nanoLang* `main()`
  - Call `nanoLang` main
  - Exit program, returning value from *nanoLang* `main()` to OS

# Ideas for Optimization

- Constant subexpression evaluation
- Common subexpression elimination
  - To do this well, we need to identify *pure* functions!
- Shift unneeded computations out of loop
- Eliminate computations of unused values

```
while(i<10)
{
        a = 3*10*i;
        b = 3*10*fun(a)+fun(a);
        i=i+1;
}
return a;
```

# Exercise: Optimization

```
Integer optFun(Integer limit)
{
    Integer sum1;
    Integer sum2;
    Integer i;
    Integer j;
    Integer a;
    Integer b;
    sum1 = 0;
    sum2 = 0;
    i=0;
    while(i<=limit)
    {
        j=0;
        while(j<=limit)
        {
            Integer c;
            a = i*i*j;
            sum1 = sum1+a;
            b=i*i;
            sum2 = sum2+b;
            c = a+b;
            j=j+1;
        }
        i=i+1;
    }
    return sum1/sum2;
}
```

End lecture 8

# Practical code generation for nanoLang

# *nanoLang* C premable (1)

```c
/*
 * Automatically generated by the nanoLang compiler ncc.
 *
 * The bolierplate and library code is released under the GNU General Public
 * Licence, version 2 or, at your choice, any later version.  Other code is
 * governed by the license of the original source code.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef long long N_Integer;
typedef char   *N_String;

#define NANO_MAKESTR(s) s
#define NANO_STRASSIGN(l, r) (l) = (r)
#define NANO_STRVAL(s) s

/* Global user variables */
```

# *nanoLang* C premable (2)

```
/* Function declarations */

N_Integer       n_Exit(N_Integer);
N_Integer       n_StrIsInt(N_String);
N_Integer       n_StrToInt(N_String);
N_Integer       n_StrLen(N_String);
N_String        n_IntToStr(N_Integer);
N_String        n_StrFront(N_String, N_Integer);
N_String        n_StrRest(N_String, N_Integer);
N_String        n_StrCat(N_String, N_String);
N_Integer       n_StrToASCII(N_String);
N_String        n_ASCIIToStr(N_Integer);
N_String        n_testfun(N_Integer, N_String);
N_Integer       n_main(N_String, N_String);

/* nanoLang runtime library code */

/* String functions */

N_String        n_StrCat(N_String arg1, N_String arg2)
{
    size_t          len = strlen(arg1) + strlen(arg2) + 1;
    char           *res = malloc(len);
    strcpy(res, arg1);
    strcat(res, arg2);
    return res;
}
[...]
```

# *nanoLang* and its translation (2)

```
String testfun(Integer count,
               String message)
{
    Integer i;
    String res;

    i=0;
    res="";

    while(i<count)
    {
        print "Schleifendurchlauf ";
        print i;
        print "\n";
        res = StrCat(res, message);
        i=i+1;
    }
    return res;
}
```

```
N_String n_testfun(N_Integer n_count,
                   N_String n_message)
{
    N_Integer       n_i = 0;
    N_String        n_res = 0;
    n_i = (0);
    NANO_STRASSIGN(n_res, (NANO_MAKESTR("")));
    while ((n_i) < (n_count)) {
        printf("%s", NANO_STRVAL((
                NANO_MAKESTR("Schleifendurchlauf "))));
        printf("%lld", (n_i));
        printf("%s", NANO_STRVAL((NANO_MAKESTR("\n"))));
        NANO_STRASSIGN(n_res, (n_StrCat((n_res),
                                        (n_message))));
        n_i = ((n_i) + (1));
    }
    return (n_res);
}
```

# *nanoLang* and its translation (2)

```
Integer main(String arg1,
             String arg2)
{
   Integer limit;
   limit = 10;

   if(StrIsInt(arg1)=1)
   {
      limit=StrToInt(arg1);
   }

   print testfun(limit, arg2);
   print "\n";

   return 0;
}
```

```
N_Integer n_main(N_String n_arg1,
                 N_String n_arg2)
{
   N_Integer       n_limit = 0;
   n_limit = (10);
   if ((n_StrIsInt((n_arg1))) == (1)) {
      n_limit = (n_StrToInt((n_arg1)));
   }
   printf("%s", NANO_STRVAL((n_testfun((n_limit),
                                (n_arg2)))));
   printf("%s", NANO_STRVAL((NANO_MAKESTR("\n"))));
   return (0);
}
```

# *nanoLang* C main

```c
/* C main function */
int main (int argc, char *argv[])
{
    N_String arg1 = NANO_MAKESTR("");
    if (1 < argc) {
        arg1 = NANO_MAKESTR(argv[1]);
    }
    N_String arg2 = NANO_MAKESTR("");
    if (2 < argc) {
        arg2 = NANO_MAKESTR(argv[2]);
    }
    n_main(arg1, arg2);
}
```

# Top-Down Parsing

# Basic Idea of *Recursive Descent*

- One parsing function per non-terminal
- Initial function corresponds to start symbol
- Each function:
  - Uses an oracle to pick the correct production
  - Processes the right hand side letter by letter against the input as follows:
    - If the next symbol of the RHS is a terminal, it consumes that terminal from the input (if it's not in the input: error)
    - If the next symbol is a non-terminal, it calls the corresponding function

Oracle: Based on next letter to be read!

- Good case: Every production can be clearly identified
- Bad case: Common initial parts of right hand sides $\Longrightarrow$ **?**

# Grammars for Recursive Descent

- A recursive descent parser deterministically finds *leftmost derivation* with a 1 token *lookahead*
- This requires an LL(1) grammar
  - No *left recursion* (results in endless descent)
  - No *shared prefixes* for rules with the same left hand non-terminal (makes it impossible to decide which rule to use)
- We can often convert a non-LL(1)-grammar to a LL(1) grammar
  - Convert left-recursion to right recursion
  - Use *left factoring* to handle common prefixes

# Example/Exercise

- Consider the following productions from $G_1$:
  - $S \to aB$
  - $B \to Sb$
  - $B \to b$
- What is the language produced?
- How can we parse $aabb$?
- What happens if we use the following productions from $G_2$?
  - $S \to aSb$
  - $S \to ab$
- Productions in $G_2$ have common prefixes
  - Common prefixes make the oracle work hard(er)
  - How can we get $G_1$ from $G_2$?

Left factoring!

# Left Factoring

- Idea: Factor common prefixes out of right hand side of productions
- Consider:
  - $N \to aR_1$
  - $N \to abR_2$
  - $N \to accR_3$
- Replace common prefix by new non-terminal:
  - $N \to aX$
  - $X \to R_1$
  - $X \to bR_2$
  - $X \to ccR_3$
- . . . and repeat as necessary

# Left recursion

**Definition (Left-recursive grammar)**

A grammar $G = \langle V_N, V_T, P, S \rangle$ is left-recursive, if there exist $A \in V_N, w \in (V_N \cup V_T)^*$ with $A \overset{+}{\Longrightarrow} Aw$.

- Left recursion leads to infinite loops in recursive descent parsers
  - To parse $A$, we first need to parse $A$ ...
- Solution: Reformulate grammar

# Our Running Example

- We will again consider the set of well-formed expressions over $x, +, *.(,)$ as an example, i.e. $L(G)$ for $G$ as follows
  - $V_N = \{E\}$
  - $V_T = \{(,), +, *, x\}$
  - Start symbol is $E$
  - Productions:
    1. $E \to x$
    2. $E \to (E)$
    3. $E \to E + E$
    4. $E \to E * E$

# Our Running Example (unambiguous)

- We will again consider the set of well-formed expressions over $x, +, *.(,)$ as an example, i.e. $L(G)$ for $G$ as follows
  - $V_N = \{E, T, F\}$
  - $V_T = \{(,), +, *, x\}$
  - Start symbol is $E$
  - Productions:
    1. $E \to E + T$
    2. $E \to T$
    3. $T \to T * F$
    4. $T \to F$
    5. $F \to (E)$
    6. $F \to x$

What happens if we want to parse $x + x$ with this grammar using recursive descent?

# Exercise: Recursive Descent for Expressions

- Consider the following productions:
  1. $E \to E + T$
  2. $E \to T$
  3. $T \to T * F$
  4. $T \to F$
  5. $F \to (E)$
  6. $F \to x$
- Can we find an equivalent grammar that can be top-down parsed?
- How?

End Lecture 9

## End of Lectures

# Eliminating Ambiguity

Ambiguous: $G = \langle V_N, V_T, P, E \rangle$

- $V_N = \{E\}$
- $V_T = \{(, ), +, *, x\}$
- $P$ as follows:
  1. $E \to x$
  2. $E \to (E)$
  3. $E \to E + E$
  4. $E \to E * E$

Unambiguous: $G' = \langle V'_N, V_T, P', E \rangle$

- $V'_N = \{E, S, F\}$
- $V_T = \{(, ), +, *, x\}$
- $P'$ as follows
  1. $E \to E + S$
  2. $E \to S$
  3. $S \to S * F$
  4. $S \to F$
  5. $F \to (E)$
  6. $F \to x$

Back

## End of Solutions

# Goals for Lecture 1

- Introduction
- Context
- Practical issues
- Programming language survey
- Execution of languages
- Low-level code vs. high-level code
- Structure of a Compiler
- Refresher
  - Grammars
  - Flex/Bison
- Programming exercises (Warmup)
  - Scientific calculator

# Exercise 0: Scientific Calculator

- Exercise 1 (Refresher):
  - Go to `http://wwwlehre.dhbw-stuttgart.de/~sschulz/cb2018.html`
  - Download `scicalcparse.y` and `scicalclex.l`
  - Build the calculator
  - Run and test the calculator
- Exercise 2 (Warm-up):
  - Add support for division and subtraction `/`, `-`
  - Add support for unary minus (the negation operator `-`)
- Exercise 3 (Bonus):
  - Change the desk calculator so that it converts its input into a C program that will perform the same actions that the calculator performed interactively!

# Review: Goals for Lecture 1

- Practical issues
- Programming language survey
- Execution of languages
- Low-level code vs. high-level code
- Structure of a Compiler
- Refresher
  - Grammars
  - Flex/Bison
- Programming exercises
  - Scientific calculator revisited

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 2

- Refresher
- Reminder: Grammars and Chomsky-Hierarchy
  - Grammars
  - Regular languages and expressions
  - Context-free grammars and languages
- Syntactic structure of programming languages
- *nanoLang*
- Programming exercise: Tokenizing *nanoLang*

# Refresher

- Some properties of programming languages and implementations
  - Object oriented vs. Procedural
  - Imperative vs. Functional
  - Statically typed vs. dynamically typed (vs. „no types“)
  - Compiled vs. interpreted
- High-level level languages
  - Expressive/Complex functionality
  - Features correspond to application concepts
  - Good abstraction
- Low-level languages
  - Simple operations
  - Features dictated by hardware architecture
  - (Close to) what processors can execute
  - Limited abstraction

# Refresher

- Structure of compiler
  - Tokenizer
  - Parser
  - Semantic analyis
  - Optimizer
  - Code generator
  - ...
- Some applications of compiler technology
  - Implementation of programming languages
  - Parsing of data formats/serialization
    - E.g. Word documents - may include optimization!
    - HTML/XML for web pages/SOA
    - XSLT document transformers
    - LaTeX
    - ATCCL
    - ...
- Flex & Bison

# Exercise 1: *nanoLang* Scanner

- Write a *flex*-based scanner for *nanoLang*
  - At minimum, it should output the program token by token
  - Bonus: Find a way to keep track of line numbers for tokens
  - Superbonus: Also keep track of columns
- Reminder: Compiling `flex` programs:

```
flex -t myflex.l > myflex.c
gcc -o myflex myflex.c
```

Example output for Hello World

```
Integer = 277
main = 274
( = 258
) = 259
{ = 270
print = 282
"Hello World\n" = 275
; = 272
return = 281
0 = 276
; = 272
} = 271
```

# Bonus Output

```
String hello(Integer dummy)
{
    return "Hello World\n";
}


Integer main()
{
    print hello(2);
    return 0;
}
```

```
<    1:  0: STRING[20]="String">
<    1:  7: IDENT[16]="hello">
<    1: 12: OPENPAR[0]="(">
<    1: 13: INTEGER[19]="Integer">
<    1: 21: IDENT[16]="dummy">
<    1: 26: CLOSEPAR[1]=")">
<    2:  0: OPENCURLY[12]="{">
<    3:  3: RETURN[23]="return">
<    3: 10: STRINGLIT[17]=""Hello World\n"">
<    3: 25: SEMICOLON[14]=";">
<    4:  0: CLOSECURLY[13]="}">
<    7:  0: INTEGER[19]="Integer">
<    7:  8: IDENT[16]="main">
<    7: 12: OPENPAR[0]="(">
<    7: 13: CLOSEPAR[1]=")">
<    8:  0: OPENCURLY[12]="{">
<    9:  3: PRINT[24]="print">
<    9:  9: IDENT[16]="hello">
<    9: 14: OPENPAR[0]="(">
<    9: 15: INTLIT[18]="2">
<    9: 16: CLOSEPAR[1]=")">
<    9: 17: SEMICOLON[14]=";">
<   10:  3: RETURN[23]="return">
<   10: 10: INTLIT[18]="0">
<   10: 11: SEMICOLON[14]=";">
<   11:  0: CLOSECURLY[13]="}">
```

# Review: Goals for Lecture 2

- Refresher
- Reminder: Grammars and Chomsky-Hierarchy
  - Grammars
  - Regular languages and expressions
  - Context-free grammars and languages
- Syntactic structure of programming languages
- *nanoLang*
- Programming exercise: Tokenizing *nanoLang*

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 3

- Refresher
- Context-free grammars revisited
  - Derivations and Parse Trees
  - Abstract Syntax Trees
- Build automation with `make`
- Flex/Bison interface
- Programming exercise: Parsing *nanoLang*

# Refresher

- Reminder: Grammars and Chomsky-Hierarchy
  - Grammars
  - Regular languages and expressions
  - Context-free grammars and languages
- Syntactic structure of programming languages
- *nanoLang*
  - Small procedural language
  - Syntax inspired by C (note: grammar has been updated!)
  - Context-free grammar
  - . . . but differences e.g. in scoping
- Programming exercise: Tokenizing nanoLang

# Exercise 2: nanoLang Parser

- Write a Bison parser for *nanoLang*
  - Bonus 1: Find out how to improve error handling (positions, recovery - try googling "Bison error handling" and "Bison error recovery" )
  - Bonus 2: Translate *nanoLang* into Abstract Syntax Trees (will be required next week!)
- Due date: Our lecture on March 5th

# Review: Goals for Lecture 3

- Refresher
- Context-free grammars revisited
  - Derivations and Parse Trees
    - Rightmost/leftmost derivation
    - Ambiguity
    - Eliminating ambiguity
  - Abstract Syntax Trees
- Build automation with `make`
- Flex/Bison interface
- Programming exercise: Parsing *nanoLang*

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 4

- Refresher
- Generating abstract syntax trees
- Parsing and error-handling
- Walk-through: Parsing *nanoLang* expressions in practice
- Programming exercise: ASTs for *nanoLang*

# Refresher

- Context-free grammars revisited
  - Derivations and Parse Trees
  - Abstract Syntax Trees
- Build automation with `make`
- Flex/Bison interface
  - `yylex()` returns numerical code for token
  - `yylex()` sets `yylval` to "semantic value"
- Programming exercise: Parsing *nanoLang*

# Exercise 3: ASTs for nanoLang

- Extend the *nanoLang* parser to generate abstract syntax trees for *nanoLang* programs
  - You can use your own parser or extend the expression parser from this lecture
  - Due date: Our lecture on April 19th

# Review: Goals for Today

- Refresher
- Generating abstract syntax trees
- Parsing and error-handling
- Walk-through: Parsing expressions in practice
- Programming exercise: ASTs for *nanoLang*

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 5

- Refresher
- Semantic properties
  - Names, variables, identifiers
  - Visibility and scopes
  - Simple types and type systems
- Symbol tables
- Memory organisation and storage locations

# Refresher

- Parse trees
- Generating abstract syntax trees
  - Leaves are delivered by the scanner
  - Internal nodes are created from subtrees in the *semantic actions* of grammar rules in Bison/YACC
- *nanoLang* expression parser
- Walk-through: Parsing expressions in practice
  - Precedence and associativity in Bison
  - AST datatype: Structure
    - Type of the AST node (roughly terminal/nonterminal of the grammar)
    - Lexeme (if needed)
    - Pointers to subtrees
- Programming exercise: ASTs for *nanoLang*

# Exercise 4: *nanoLang* Types and Symbol Tables

- Develop data structures for representing *nanoLang* types
- Develop data structures for implementing nested symbol tables
- Due date: Our lecture on April 19th

# Review: Goals for Today

- Semantic properties
  - Names, variables, identifiers
  - Visibility and scopes
  - Simple types and type systems
- Symbol tables
- Memory organisation and storage locations

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 6

- Refresher
- The big picture
- Symbol Tables and Type Tables in practice
- Type inference and type checking
- Exercise: Build symbol tables

# Refresher

- Semantic properties
  - Names, variables, identifiers
  - Visibility and scopes
  - Simple types and type systems
- Symbol tables
- Memory organisation and storage locations
  - Global variabes: data segment
  - Local ("automatic") variables: calling stack
  - Dynamic memory (managed): heap

Lecture 6

# Exercise 5: Implementing Symbol Tables

Extend your compiler project by computing the relevant symbol tables for all nodes of your AST

- Develop a type table date type for managing different types
- Define a symbol table data type for managing symbols and their types
  - Use a hierarchical structure
  - Suggested operations:
    - `EnterScope()`
    - `LeaveScope()`
    - `InserSymbol()` (with type)
    - `FindSymbol()` (return entry including type)
    - ...
- Traverse the AST in a top-down fashion, computing the valid symbol table at each node
- Annotate each AST node with the symbol table valid at that node

At the end, print all symbols and types of the global, top-level symbol table!

# Example Output

```
> ncc NANOEXAMPLES/scopes.nano
Global symbols:
---------------
i                       : Integer
fun1                    : (Integer) -> Integer
main                    : () -> Integer

Types:
------
0: NoType
1: String
2: Integer
3: (Integer) -> Integer
4: () -> Integer
```

# Review: Goals for Lecture 6

- Refresher
- The big picture
  1. We need to type every expression and subexpression
- Symbol Tables and Type Tables in practice
  1. Build tables top-down
  2. Annotate AST nodes with symbols tables
- Type inference and type checking
  1. Bottom-up type inference
  2. Check constraints as needed
  3. In more complex languages (C)
     - More complex type inference
     - Type promotion
- Exercise: Build symbol tables

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 7

- Refresher
- Type checking revisited
- Excursion: `assert()` in C
- Code generation considerations
  - Parameter passing
  - Assignments
  - Calling conventions
  - Runtime support
  - Libraries
- *nanoLang* runtime
  - Parameter passing
  - *nanoLang* string semantics
  - *nanoLang* library functions and OS interaction
- Exercise: Type checking

# Refresher

- The big picture
  1. We need to type every expression and subexpression
- Symbol Tables and Type Tables in practice
  1. Build tables top-down
  2. Annotate AST nodes with symbols tables
- Type inference and type checking
  1. Bottom-up type inference
  2. Check constraints as needed
  3. In more complex languages (C): type promotion

Lecture 7

# Exercise 6: Typechecking Nanolang

Extend your compiler project by computing the types of all expressions in your system and check type constraints

- Check that variables are only assigned values of the right type
- Check that functions are only called with correctly typed parameters
- Check that operators have compatible types
- Check that comparisons only happen between expressions of the same type
- Bonus: Check that functions (always) return the correct type

# Exercise 6: Example Typechecking

```
>  ncc ../NANOEXAMPLES/semantic1.nano
1:25: error: symbol 'i' doubly defined (previous
      definition at 1:14)
7:13: error: undefined identifier j
15:4: error: undefined identifier fun2
15:4: error: undefined identifier fun2
1:1: warning: cannot guarantee proper
     return value for function fun1()
12:1: warning: cannot guarantee proper
      return value for function main()
```

# Review: Goals for Lecture 7

- Type checking revisited
- Excursion: `assert()` in C
- Code generation considerations
  - Parameter passing
  - Assignments
  - Calling conventions
  - Runtime support
  - Libraries
- *nanoLang* runtime
  - Parameter passing
  - *nanoLang* string semantics
  - *nanoLang* library functions and OS interaction

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 8

- Refresher
- Coding Hints
- Code generation *nanoLang* to C
- (Simple) Optimizations
- Exercise: Code generation (I)

# Refresher

- Type checking revisited
- Excursion: `assert()` in C
- Code generation considerations
  - Parameter passing
  - Assignments
  - Calling conventions
  - Runtime support
  - Libraries
- *nanoLang* runtime
  - Parameter passing
  - *nanoLang* string semantics
  - *nanoLang* library functions and OS interaction

Lecture 8

# Exercise 7: Basic Code Generation

Extend your compiler project to generate a basic C programm

- Compile *nanoLang* statements into equivalent C statements
- Compile `nanoLang` definitions into C declarations and definitions
- Generate a basic `main()`
- For now, you can treat `String` as an imutable `char*` - we'll do the library next week

# Review: Goals for Lecture 8

- Coding Hints
- Code generation *nanoLang* to C
  - (Name mangling)
  - Boilerplate
  - Globals
  - Library code
  - *nanoLang* functions
  - C `main()`
- (Simple) Optimizations

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 9

- Refresher
- Practical aspects of *nanoLang* code generation
- An introduction to top-down recursive descent parsing

# Refresher

- Coding Hints
  - Code structure (small functions with one clear job)
  - Reasonable comments
  - `assert()` and/or unit tests
- Code generation *nanoLang* to C
  - Preamble
  - Mine symbol tables for declarations
  - Library
  - Walk the AST to generate actual code
- (Simple) Optimizations
  - Constant subexpressions
  - Common subexpressions
  - Lift code out of loops
  - Detect and remove unused code and variables
- Exercise: Code generation (I)

# Exercise 8: Completing the Compiler

- Finish your compiler project
- Bonus: Implement proper string handling
  - String data type
  - Garbage collection

# Review: Goals for Lecture 9

- Practical aspects of *nanoLang* code generation
- An introduction to top-down recursive descent parsing

# Feedback round

- What was the best part of today's lecture?
- What part of today's lecture has the most potential for improvement?
  - Optional: how would you improve it?

# Goals for Lecture 10

- Training exam
- Solution discussion

# Training exam

# Review: Goals for Lecture 10

- Training exam
- Solution discussion

# Feedback round

- What was the best part of the course?
- Suggestions for improvements?

# The End