Reminder: How Does a Computer Run Your C Program?

- You edit `myprogram.c`
- You compile: `cc -o myprogram myprogram.c`
  - Preprocessor generates `preprocessed source` (`myprogram.i`)
  - Compiler proper generates `assembly program` (`myprogram.s`)
  - Assembler generates `object code` (`myprogram.o`)
  - Linker generates `executable` (`myprogram`)
- You “run” it: `./myprogram`
  - Operating system generates a new process
  - Dynamic linker resolves references to shared libraries
  - Loader generates `executable in-memory image`
  - CPU runs machine code

Stages in Translating a Program

Lexical analysis (Scanner): Breaking a program into primitive components, called tokens (identifiers, numbers, keywords, …)

Syntactic analysis (Parsing): Creating a syntax tree of the program.
  - Symbol table: Storing information about declared objects (identifiers, procedure names, …)

Semantic analysis: Understanding the relationship among the tokens in the program.

Optimization: Rewriting the syntax tree to create a more efficient program.

Code generation: Converting the parsed program into an executable form.

Each stage is based on a specification of the relevant language aspect!

Programming Language Implementation

Translation

Source language programs are translated into target language programs:
  - Assembler: symbolic representation of machine code → machine code
  - Compiler: high(er)-level language → low(er)-level language
  - Loader / link editor translates address references in object code indicated by address tables to actual addresses
  - Macroprocessor / preprocessor performs macro expansion and code fragment selection by applying rewriting rules

Description of Programming Languages

Syntax — Shape of PL constructs
- What are the tokens of the language? — Lexical syntax, “word level”
- How are programs built from tokens? — Mostly use Context-Free Grammars (CFG) or Backus-Naur-Form (BNF) to describe syntax at the “sentence level”
- Which further constraints are there on program structure? — “Static semantics”: aspects of program structure that are checked at compile time, but cannot be captured by CFGs ( → context-sensitive syntax):
  - Scopes of names
  - Typing

Semantics — Meaning of PL constructs
Three major approaches to PL semantics:
- Axiomatic semantics: `{p \rightarrow \text{Prog}}{q}`
- Denotational semantics: \text{Prog} denotes a mathematical function \([\text{Prog}]\)
- Operational semantics: state transition sequence of an abstract machine
Formal Languages, Grammars, Automata

A **formal language** over an alphabet $A$ is a subset of $A^*$. Formal languages can be **generated** by grammars, **recognized** by automata.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input Alphabet</th>
<th>Output</th>
<th>Grammar Type</th>
<th>Recognising Automata</th>
<th>Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexing</td>
<td>Characters</td>
<td>Token Sequence</td>
<td>Type 3: Regular</td>
<td>Finite Automata</td>
<td>lex, flex, ocamllex, alex</td>
</tr>
<tr>
<td>Parsing</td>
<td>Tokens</td>
<td>Syntax Tree</td>
<td>Type 2: Context-Free</td>
<td>Pushdown Automata</td>
<td>yacc, bison, ANTLR, JavaCC, ocaml, yacc, happy</td>
</tr>
</tbody>
</table>

Two levels of formal languages:
- **token languages** over character-level alphabet
- **program language** over token alphabet

---

Token Example: Identifiers in Java

Java 2 Language Spec. 3.8:

**IdentifierChars:**

```
JavaLetter
IdentifierChars JavaLetterOrDigit
```

Conventional BNF:

```
IdentifierChars ::= JavaLetter
                | IdentifierChars JavaLetterOrDigit
```

Conventional CFG:

```
IdentifierChars → JavaLetter
IdentifierChars → IdentifierChars JavaLetterOrDigit
```

“Railroad diagram”:

```
IdentifierChars = JavaLetter + JavaLetterOrDigit
```

Regular Expression:

```
IdentifierChars = JavaLetter · JavaLetterOrDigit *
```

---

Lexical Analysis

- Lexical syntax is defined as a set of **token classes**
- Lexical analysis: find out which token class contains a prefix of the character stream
- Each token class corresponds to a regular language (typically all disjoint)
- **Regular languages** are
  - the languages **generated by regular grammars**,
  - the languages **accepted by finite-state automata**,
  - the languages **denoted by regular expressions**.

---

Regular Expressions

**Definition:** A regular expression over an alphabet $\Sigma$ is

- $\varepsilon$, standing for the empty string
- an element of $\Sigma$
- alternative $M | N$ of two regular expressions $M$ and $N$
- concatenation $MN$ of two regular expressions $M$ and $N$
- iteration $M^*$ of a regular expressions $M$

Each regular expression denotes a **regular language**:

- $\varepsilon = \{\varepsilon\}$
- If $a \in \Sigma$, then $a = \{a\}$
- $M | N = M \cup N$ — union of languages
- $MN = M \cdot N$ — concatenation of languages
- $M^* = \bigcup_{i \in \mathbb{N}} M^i$
Regular Expressions — Rigorous Version

**Definition:** The set of regular expressions over an alphabet \( \Sigma \) is the smallest set such that:
- \( \varepsilon \) (standing for the empty string) is a regular expression
- \( a \) is a regular expression for each \( a \in \Sigma \),
- for any two regular expressions \( M \) and \( N \), their alternative \( M \mid N \) is a regular expression
- for any two regular expressions \( M \) and \( N \), their concatenation \( MN \) is a regular expression
- for any regular expression \( M \), its iteration \( M^* \) is a regular expression

Each regular expression \( M \) over \( \Sigma \) denotes a regular language \([ M ] : \mathbb{P} \Sigma^*:\)
- \([ \varepsilon ] = \{ \varepsilon \}\)
- If \( a \in \Sigma \), then \([ a ] = \{ a \}\)
- If \( M \) and \( N \) are regular expressions, then \([ M \mid N ] = [ M ] \cup [ N ] \) — union of languages
- If \( M \) and \( N \) are regular expressions, then \([ MN ] = [ M ] \cdot [ N ] \) — concatenation of languages
- If \( M \) is a regular expression, then \([ M^* ] = \bigcup_{i \in \mathbb{N}} [ M ]^i \)

**Extended Regular Expressions**

- \( M^+ \equiv MM^* = \bigcup_{i \in \mathbb{N}} \setminus \{ 0 \} \)
- \( M? \equiv M \mid \varepsilon \)
- \([ a-z] \equiv a \mid b \mid c \mid \cdots \mid y \mid z \) — requires a linear ordering on \( \Sigma \)
- \([ a-zA-Z] \equiv [ a-z] \mid [ A-Z] \)
- \( = \Sigma \)
- \([ ^a-z] = \Sigma \setminus [a-z] \)

- Read the UNIX manual pages for **grep** and **egrep**: compare the regular expressions there with those here and with those in the textbook.
- Learn what **awk** and **sed** are used for (UNIX texts, manual pages), and what the basic structure of **awk** and **sed** scripts is.
- Have you ever encountered any problems that you now would solve using **grep**, **awk**, and **sed**?

**Regular Expression Examples**

- \( Nat = [0-9]^+ \)
- \( Integer = -?[0-9]^+ \)
- \( Identifier = [a-zA-Z][a-zA-Z0-9]^* \)
- \( LineComment = /[^\v\n\f]+[^\v\n\f] / \)

**Lexer Generators** convert regular expression token definitions into efficient implementations of finite-state automata
- **lex, flex, Jlex, Alex, ocamllex**...

**Lexer Generation for C — flex**

- Original AT&T UNIX: **lex**
- GNU re-implementation: **flex**
- File naming convention: *.l → lex.yy.c
- **Rules:** actions guarded by regular expression patterns
- Generates automata-based stream processors

```c
/* user.l */

%option outfile="user.c"
%option main
%
userID printf( "%d", getuid() );
```
Small flex Documentation Example (adapted)

```c
%option outfile="count.c"
%option noyywrap
/* so we don’t need “-lfl” */
{%
    int num_lines = 0, num_chars = 0;
%
%

\n++num_lines; ++num_chars;
.
++num_chars;
%
%
int main()
{
    yylex();
    printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars );
    return 0;
}
```
**Derivations and Parse Trees**

Integer → Integer Digit → Integer Digit Digit
→ Digit Digit Digit → 3 Digit Digit → 35 Digit → 352

Integer → Integer Digit → Integer 2
→ Integer Digit 2 → Integer 52 → Digit 52 → 352

**Regular Grammars**

If all productions are of shape $N_1 \rightarrow t$ or $N_1 \rightarrow N_2 t$, then the grammar is called **regular**.

InputElement → WhiteSpace | Token
WhiteSpace → ' ' | \n | \v | \y | \v\n
Token → Identifier | Number | Separator
Identifier → Letter | Identifier Letter | Identifier Digit
Number → Digit | Number Digit
Letter → a | b | ... | z | A | B | ... | Z
Digit → 0 | 1 | ... | 9
Separator → ( | ) | { | } | ; | ,

**Regular Grammars — Simplified Example**

InputElement → WhiteSpace | Token
WhiteSpace → ’ ’ | \v | \n | \y | \v\n
Token → Identifier | Number | Separator
Identifier → Letter | Identifier Letter | Identifier Digit
Number → Digit | Number Digit
Letter → a | b | ... | z | A | B | ... | Z
Digit → 0 | 1 | ... | 9
Separator → ( | ) | { | } | ; | ,

**Regular Expressions vs. Context-Free Grammars**

**Definition:** A **regular expression** over an alphabet $\Sigma$ is
- $\varepsilon$, standing for the empty string
- an element of $\Sigma$
- alternative $M | N$ of two regular expressions $M$ and $N$
- concatenation $MN$ of two regular expressions $M$ and $N$
- iteration $M^*$ of a regular expressions $M$

**Definition:** A **context-free grammar (CFG)** is a tuple $(\Sigma, N, S, \rho)$ where
- $\Sigma$ is a set of **terminal symbols**
- $N$ is a set of **nonterminal symbols**
- $S \in N$ is the **start symbol**
- $\rho \subseteq (N \times (N \cup \Sigma)^*)$ is a set of rules;
- a rule $(A, \omega)$ is usually written “$A \rightarrow \omega$”
Regular Languages vs. Context-Free Languages

A language is **regular** iff there is a regular expression denoting it
- **Fact:** A language is regular iff there is a DFA accepting it
- **Fact:** A language is regular iff there is a NFA accepting it

A language is **context-free** iff there is a context-free grammar generating it
- **Fact:** A language is context-free iff there is a pushdown-automaton (= NFA with stack) accepting it
- **Fact:** All regular languages are context-free
- **Fact:** Many context-free languages are **not** regular

Examples:
- \( \{ a^n b^n \} = \bigcup_{n \in \mathbb{N}} a^n b^n \)
- Expression languages with matching parentheses nested to arbitrary depth
- Palindromes

Ambiguity

\[
\text{Exp} \rightarrow \text{Integer} \mid \text{Exp} + \text{Exp} \mid \text{Exp} - \text{Exp} \mid \text{Exp} \times \text{Exp} \mid \text{Exp} / \text{Exp}
\]

Programming language grammars should not be ambiguous!

48 / 6 / 2

Concrete Syntax of Arithmetic Expressions

We need a grammar with the following requirements:
- unambiguous parse for each syntactically correct expression
- parse trees reflect expression structure
- parentheses are input symbols

For reference: The **abstract syntax grammar** again:

\[
\text{Expression} \rightarrow \text{Literal} \\
\mid \text{Identifier} \\
\mid \text{Expression} + \text{Expression} \\
\mid \text{Expression} - \text{Expression} \\
\mid \text{Expression} \times \text{Expression} \\
\mid \text{Expression} / \text{Expression}
\]

An **expression** can be
- a number literal
- a variable
- an application of a binary operator (+,-,\({}\),\(/\)) to two expressions

**Abstract syntax grammars are tree grammars!**
A Grammar for Concrete Syntax of Arithmetic Expressions

\[ \text{Expr} \rightarrow \text{Term} \]
\[ \mid \text{Expr} + \text{Term} \]
\[ \mid \text{Expr} - \text{Term} \]

\[ \text{Term} \rightarrow \text{Factor} \]
\[ \mid \text{Term} \ast \text{Factor} \]
\[ \mid \text{Term} / \text{Factor} \]

\[ \text{Factor} \rightarrow \text{Ident} \]
\[ \mid \text{Literal} \]
\[ \mid ( \text{Expr} ) \]

Expression Datatype in Java

abstract class Expr { // Expr = Value + Variable + Binary

class Value extends Expr { // Value = int
    int intValue;
}

class Variable extends Expr { // Variable = String (intended)
    String name;
}

class Binary extends Expr { // Binary = Expr \times Operator \times Expr
    Operator op;
    Expr term1, term2;
}

How do we implement alternatives like Value + Variable + Binary in C?

Interlude — Union Datatypes in C

```c
#include <stdio.h> // Union.c
#include <string.h>

typedef union {
    char name[8];
    double value;
    int rank;
} MyUnion;

int main ( int argc, char * argv[] ) {
    MyUnion u;
    strncpy( u.name, argc > 1 ? argv[1] : "McMaster", 8);
    printf("size=%d\nvalue=%g\nrank=%d\n", sizeof( u ), u.value, u.rank);
}
```

- Syntax like struct
- All components are located at the same address
- unions should only be used tagged!

Expression Datatype in C — Prelude

```c
#include <stdlib.h> // Expr.c
#include <stdio.h>
#include <string.h>
```

How do we implement alternatives like Value + Variable + Binary in C?
Expression Datatype in C

typedef struct ExprStruct * Expr;

typedef struct {
    Expr left;
    char op[4]; // only short operators!
    Expr right;
} BinRec;

typedef enum { tagNum, tagVar, tagBin } Tag;

struct ExprStruct { // record containing tagged union
    Tag tag;
    union {
        long int num; // for tagNum
        char * name; // for tagVar
        BinRec bin; // for tagBin
    } u;
} // Note the struct field label “u”

Expression Datatype in C — Interface

// pointer types need not have declared destination struct type!
typedef struct ExprStruct * Expr; // Expr.h

extern Expr exprInt(long int n); // Expr.h
extern Expr exprVar(char * ident);
extern Expr exprBin(char * op, Expr e1, Expr e2);
extern long int exprEval(Expr e);

The implementation is completely hidden!
⇒ Look Ma, no union!

Literal Expression Construction in C

Expr exprInt(long int n) {
    Expr result = malloc(sizeof(struct ExprStruct));
    if ( result == NULL) return NULL;
    result->tag = tagNum;
    result->u.num = n;
    return result;
}

Expr exprVar(char * ident) {
    Expr result = malloc(sizeof(struct ExprStruct));
    if ( result == NULL) return NULL;
    result->tag = tagVar;
    result->u.name = strdup(ident);
    return result;
}

Expr exprBin(char * op, Expr e1, Expr e2) {  
    if ( op == NULL || strlen(op) > 3 ) return NULL;
    Expr result = malloc(sizeof(struct ExprStruct));
    if ( result == NULL ) return NULL;
    result->tag = tagBin;
    result->u.bin.left = e1;
    result->u.bin.right = e2;
    strcpy(result->u.bin.op, op);
    return result;
}
Naïve Evaluation of Ground Expressions in C

long int exprEval(Expr e) {
    switch (e→tag) {
        case tagNum: return e→u.num;
        case tagBin: {
            long int val1 = exprEval(e→u.bin.left);
            long int val2 = exprEval(e→u.bin.right);
            switch (e→u.bin.op[0]) { // only for demonstration!
                case '+': return val1 + val2;
                case '-': return val1 - val2;
                case '*': return val1 * val2;
                case '/': return val1 / val2;
                default: fprintf(stderr,"exprEval: illegal operator '%s'\n", e→u.bin.op);
            }
        }
        case tagVar: fprintf(stderr,"exprEval: unexpected variable '%s'\n", e→u.name);
        break;
        default: fprintf(stderr,"exprEval: illegal tag\n");
    }
    exit(1);   // all error exit goes through this
}

Expression Test

#include <stdio.h>
#include "Expr.h"

int main ( void ) {
    Expr e6 = exprInt(6);
    Expr e7 = exprInt(7);
    Expr answer = exprBin("*", e6, e7);
    printf("answer = %ld\n", exprEval( answer ));
    printf(" ... %ld\n", exprEval( exprBin("+", answer, exprInt(14)) ));
    Expr eX = exprVar("x");
    printf("answer = %ld\n", exprEval( exprBin("-", answer, eX ) ));
    return 0;
}

Expression Parsing Examples

Expression → Term | Expression + Term | Expression - Term
Term → Factor | Term * Factor | Term / Factor
Factor → Ident | Literal | ( Expression )

* Parsing "\((a + b) * c\)"

48 / y + x  48 / (y + x)  48 / (y / x)  48 / y / x  (48 / y) / x
Dangling else

IfStatement → if ( Expression ) Statement
|  if ( Expression ) Statement else Statement

if( x<0 ) if( y<0 ) y=y-1; else y=0;  if( x<0 ) if( y<0 ) y=y-1; else y=0;

Solutions:
• non-CFG rules — C, C++
• extra non-terminal StatementNoShortIf — Java
• end if, endif, fi — Ada
• no “short if” — Haskell

Parser Generation Using bison

• Original version: AT&T UNIX yacc — “yet another compiler compiler”
• GNU version: bison
  – Backward-compatible: bison -y (produces y.tab.c)
  – Extensions, including GLR parsing — arbitrary CFGs
• Rules are grammar productions with semantic actions
• General flavour of “semantic actions” is functional:
  – defining the value $$ of the currently recognised structure
  – in terms of the values $1, $2, … of its constituents
• Special support for semantics as union types
• bison –d produces token definition file for lexer
• Semantic types are shared with lexer

flex Lexer Returning Tokens

%option noyywrap outfile="simple Lexer.c"
/* scanner for a toy calculator */
simple Lexer.l */
%
#include "Expr.h"  /* required for the types in next line */
#include "simple Parser.tab.h"  /* token definitions and types */
%
%%
[0-9]+  yylval intval = atoi(yytext); return TOK_NUMBER;
if  return TOK_IF;
then  return TOK_THEN;
else  return TOK_ELSE;
[a-z][a-z0-9]*  yylval string = strdup(yytext); return TOK_ID;
[ \t]+  /* eat up whitespace */
[\s-\v()]\n  { return yytext[0];}
    .  fprintf(stderr, "Unrecognized character: %s\n", yytext ); return −1;

Simple Expression Parser

%
#include <stdio.h>
#include "Expr.h"
int yylex(void);
void yyerror (char const * s) { fprintf(stderr, "%s\n", s); }
%
%union {
  long int intval;
  char * string;
  Expr expr;
}
%token <intval> TOK_NUMBER
%token <string> TOK_ID
%token TOK_IF TOK_THEN TOK_ELSE
%type <EXPR> expr term factor
%start input
%%
The Language “Make”

- **Rule-based** artefact production language
- Rules (normally) specify how to produce **targets** from their prerequisites
- Rules consist of a description of a **dependency relation** and and **action**
- A **make** run performs a bottom-up traversal of the dependency tree
- Actions are triggered unless a target is newer than all its prerequisites
- Actions are specified in a shell language (sh, bash)
- Performing the action of a rule should satisfy its dependency
- **make** and in particular **gmake** has a wealth of **built-in rules** and default definitions
- Actions are introduced by **leading TAB characters**

Modified Exercise 2.3

- ... 
- Further modify the simple calculator presented in class so that it accepts definitions of variables, introduced by the keyword “let”:

```plaintext
let x = 4
let y = 5
x+y
= 9
```
- Further modify the simple calculator presented in class so that it produces step-wise evaluation traces:

```plaintext
(4+3) * 8 - 2*7
= (4 + 3) * 8 - 2 * 7
= 7 * 8 - 2 * 7
= 56 - 2 * 7
= 56 - 14
= 42
```