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

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

Software simulation — Virtual machines

Create a (low-level) program that acts as a “computer whose machine language is the high-level language”.

This **interpreter** also acts as a **virtual machine** implementation.

Most “interpreters” first perform compilation into some internal representation (sometimes exported as **bytecode**).

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!

Describing 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:** $\{p\}\ \texttt{Prog}\ \{q\}$
- **Denotational semantics:** `Prog` *denotes* a *mathematical* function $[\![\texttt{Prog}]\!]$
- **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**.

Phase	*Input Alphabet*	*Output*	*Grammar Type*	*Recognising Automata*	*Generators*

Lexing	Characters	Token **Sequence**	Type 3: Regular	Finite Automata	`lex, flex`
					`ocamllex`
					`alex`

Parsing	Tokens	Syntax **Tree**	Type 2: Context-Free	Pushdown Automata	`yacc, bison`
					`ANTLR, JavaCC`
					`ocamlyacc,`
					`happy`

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
```

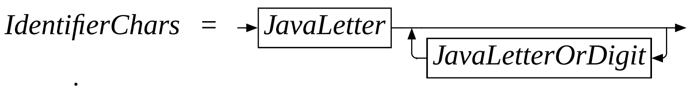
Conventional BNF:

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

Conventional CFG:

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

“Railroad diagram”:



Regular Expression:

$$\mathtt{IdentifierChars} = \mathtt{JavaLetter} \cdot \mathtt{JavaLetterOrDigit}^*$$

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 Σ is

- ε, standing for the empty string
- an element of Σ
- alternative $M \mid N$ 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**:

- $\varepsilon = \{\langle\rangle\}$
- If $a \in \Sigma$, then $a = \{\langle a \rangle\}$
- $M \mid N = M \cup N$ — 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** Σ is the smallest set such that:

- ε (standing for the empty string) is a regular expression
- $\underline{a}$ 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 Σ ***denotes*** a regular language $\llbracket M \rrbracket : \mathbb{P}\ \Sigma^*$:

- $\llbracket \varepsilon \rrbracket = \{\langle\rangle\}$
- If $a \in \Sigma$, then $\llbracket \underline{a} \rrbracket = \{\langle a \rangle\}$
- If M and N are regular expressions, then $\llbracket M \mid N \rrbracket = \llbracket M \rrbracket \cup \llbracket N \rrbracket$ — union of languages
- If M and N are regular expressions, then $\llbracket MN \rrbracket = \llbracket M \rrbracket \cdot \llbracket N \rrbracket$ — concatenation of languages
- If M is a regular expression, then $\llbracket M^* \rrbracket = \bigcup_{i \in \mathbb{N}} \llbracket M \rrbracket^i$

Extended Regular Expressions

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

- 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

- $\mathit{Nat} = [0\text{-}9]^+$
- $\mathit{Integer} = \text{-}?[0\text{-}9]^+$
- $\mathit{Identifier} = [a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*$
- $\mathit{LineComment} = /\!/[\hat{}\backslash r\backslash n\backslash f]^*[\backslash r\backslash n\backslash f]$

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

```
                    /* user.l */
%option outfile="user.c"
%option main
%%
userID    printf ( "%d", getuid() );
```

Small flex Documentation Example (adapted)

```
%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;
}
```

Larger flex Documentation Example (adapted)

```
%option noyywrap outfile="toy_lexer.c"
/* scanner for a toy Pascal-like language                              toy.l  */
%{
#include <math.h>          /* need this for the call to atof() below */
%}
DIGIT    [0–9]
ID       [a–z][a–z0–9]*
%%
{DIGIT}+                          printf( "An integer: %s (%d)\n", yytext, atoi( yytext ) );
{DIGIT}+"."{DIGIT}*               printf( "A float: %s (%g)\n", yytext, atof( yytext ) );
if|then|begin|end|procedure|function      printf( "A keyword: %s\n", yytext );
{ID}                              printf( "An identifier: %s\n", yytext );
"+"|"-"|"*"|"/"                   printf( "An operator: %s\n", yytext );
"{"[^{}\n]*"}"                    /* eat up one-line comments */
[ \t\n]+                          /* eat up whitespace */
.                                 printf( "Unrecognized character: %s\n", yytext );
%%
```

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

BNF in the Textbook

$$
\begin{array}{lcl}
\mathtt{Integer} & \to & \mathtt{Digit} \mid \mathtt{Integer\ Digit} \\
\mathtt{Digit} & \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

This is an abbreviation for the following set of CFG rules:

$$
\begin{array}{lcl}
\mathtt{Integer} & \to & \mathtt{Digit} \\
\mathtt{Integer} & \to & \mathtt{Integer\ Digit} \\
\mathtt{Digit} & \to & 0 \\
\mathtt{Digit} & \to & 1 \\
 & \vdots & \\
\mathtt{Digit} & \to & 9
\end{array}
$$

Definition: A **context-free grammar (CFG)** is a tuple (Σ, N, S, ρ) where

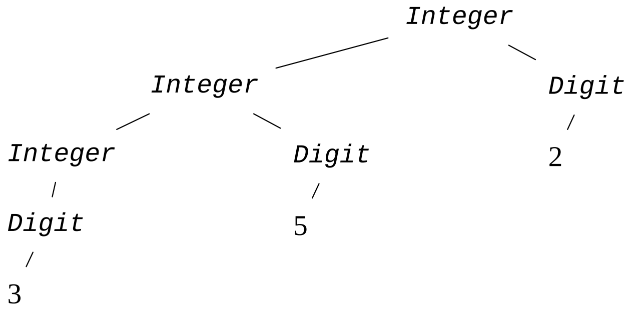
- Σ is a set of **terminal symbols**
- N is a set of **nonterminal symbols**
- $S \in N$ is the **start** symbol
- $\rho \subseteq (N \times (N \cup \Sigma)^*)$ is a set of rules;

  a rule (A, ω) is usually written “$A \to \omega$”

Derivations and Parse Trees

$$\texttt{Integer} \rightarrow \texttt{Integer Digit} \rightarrow \texttt{Integer Digit Digit}$$
$$\rightarrow \texttt{Digit Digit Digit} \rightarrow 3\,\texttt{Digit Digit} \rightarrow 35\,\texttt{Digit} \rightarrow 352$$

$$\texttt{Integer} \rightarrow \texttt{Integer Digit} \rightarrow \texttt{Integer}\,2$$
$$\rightarrow \texttt{Integer Digit}\,2 \rightarrow \texttt{Integer}\,52 \rightarrow \texttt{Digit}\,52 \rightarrow 352$$



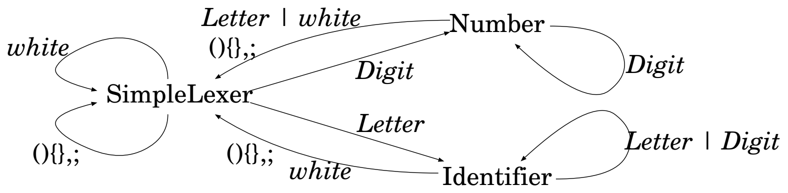
Regular Grammars

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

InputElement	$\rightarrow$	*WhiteSpace* | *Comment* | *Token*
WhiteSpace	$\rightarrow$	' ' | \t | \r | \n | \f | \r\n
Token	$\rightarrow$	*Identifier* | *Keyword* | *Literal* | *Separator* | *Operator*
Identifier	$\rightarrow$	*Letter* | *Identifier Letter* | *Identifier Digit*
Letter	$\rightarrow$	`a` | `b` | … | `z` | `A` | `B` | … | `Z`
Digit	$\rightarrow$	0 | 1 | … | 9
Keyword	$\rightarrow$	`boolean` | `else` | `if` | `int` | `main` | `void` | `while`
Separator	$\rightarrow$	`(` | `)` | `{` | `}` | `;` | `,`
	⋮	

Regular Grammars — Simplified Example

InputElement	$\rightarrow$	*WhiteSpace* | *Token*
WhiteSpace	$\rightarrow$	' ' | \t | \r | \n | \f | \r\n
Token	$\rightarrow$	*Identifier* | *Number* | *Separator*
Identifier	$\rightarrow$	*Letter* | *Identifier Letter* | *Identifier Digit*
Number	$\rightarrow$	*Digit* | *Number Digit*
Letter	$\rightarrow$	`a` | `b` | … | `z` | `A` | `B` | … | `Z`
Digit	$\rightarrow$	0 | 1 | … | 9
Separator	$\rightarrow$	`(` | `)` | `{` | `}` | `;` | `,`



Regular Expressions vs. Context-Free Grammars

Definition: A **regular expression** over an alphabet Σ is

- ε, standing for the empty string
- an element of Σ
- alternative $M \mid N$ 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 (Σ, N, S, ρ) where

- Σ 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, ω) 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 lanugages are context-free**
- ***Fact:*** **Many context-free languages are not regular**

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

Ambiguity

$$Exp \rightarrow Integer \;\mid\; Exp + Exp \;\mid\; Exp - Exp \;\mid\; Exp * Exp \;\mid\; Exp / Exp$$

$48 \;/\; 6 \;/\; 2$ $\qquad\qquad$ $48 \;/\; 6 \;/\; 2$

Programming language grammars should not be ambiguous!

Abstract Syntax of Arithmetic Expressions

$$\begin{array}{rl} Expression \rightarrow & Literal \\ \mid & Identifier \\ \mid & Expression \;\; \texttt{+} \;\; Expression \\ \mid & Expression \;\; - \;\; Expression \\ \mid & Expression \;\; * \;\; Expression \\ \mid & Expression \,/\, Expression \end{array}$$

An **expression** can be

- a number literal
- a variable
- an application of a binary operator (+,-,*,/) to two expressions

Abstract syntax grammars are *tree grammars!*

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 **abtract syntax grammar** again:

$$\begin{array}{rl} Expression \rightarrow & Literal \\ \mid & Identifier \\ \mid & Expression \;\; \texttt{+} \;\; Expression \\ \mid & Expression \;\; - \;\; Expression \\ \mid & Expression \;\; * \;\; Expression \\ \mid & Expression \;\; / \;\; Expression \end{array}$$

A Grammar for Concrete Syntax of Arithmetic Expressions

$$\begin{aligned}
Expr &\to Term \\
&\mid Expr \;\; \mathtt{+} \;\; Term \\
&\mid Expr \;\; \mathtt{-} \;\; Term \\
Term &\to Factor \\
&\mid Term \;\; \ast \;\; Factor \\
&\mid Term \;\; \mathtt{/} \;\; Factor \\
Factor &\to Ident \\
&\mid Literal \\
&\mid \mathtt{(} \;\; Expr \;\; \mathtt{)}
\end{aligned}$$

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 × Operator × Expr
  Operator op;
  Expr term1, term2;
}
```

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

Interlude — Union Datatypes in 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

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

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);
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;
}
```

- *NULL* return value as failure indicator
- *Expr* **datatype invariant:**

  While *e*→*tag* == *tagNum*, only the *num* field of *e*→*u* may be accessed!

Expression Construction in C

```
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);
           }}
           break;
    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
```

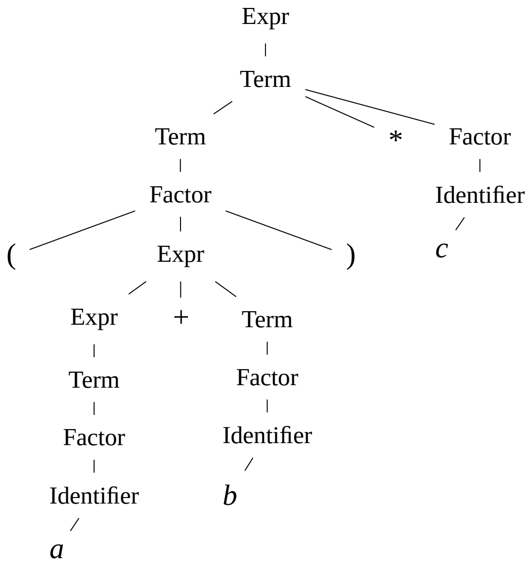
Expression Test

```
#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;
}
```

Parsing “$(a + b) * c$”

$$\begin{aligned} Expr &\to Term \\ &\mid Expr + Term \\ &\mid Expr - Term \\ Term &\to Factor \\ &\mid Term * Factor \\ &\mid Term \,/\, Factor \\ Factor &\to Ident \\ &\mid Literal \\ &\mid (\ Expr\ ) \end{aligned}$$



Expression Parsing Examples

$$\begin{aligned} Expression &\to Term \mid Expression + Term \mid Expression - Term \\ Term &\to Factor \mid Term * Factor \mid Term \,/\, Factor \\ Factor &\to Identifier \mid Literal \mid (\ Expression\ ) \end{aligned}$$

$48 / y + x$ $48 / (y + x)$ $48 / (y / x)$ $48 / y / x$ $(48 / y) / x$

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 */
[+\−∗/()\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
%%
```

```
factor : TOK_NUMBER        { $$ = exprInt($1); }
       | TOK_ID            { $$ = exprVar($1); }
       | ’(’ expr ’)’      { $$ = $2; }

term : factor
     | term ’*’ factor  { $$ = exprBin("*", $1, $3); }
     | term ’/’ factor  { $$ = exprBin("/", $1, $3); }

expr : term
     | expr ’+’ term  { $$ = exprBin("+", $1, $3); }
     | expr ’-’ term  { $$ = exprBin("-", $1, $3); }

input : /* empty */ | input line        /* line-by-line processing */

line : ’\n’            {}               /* empty lines allowed */
     | expr ’\n’       { printf(" = %ld\n", exprEval($1)); }
%%
int main ( void ) { return yyparse(); }
```

Makefile

```
simple_calc: simple_parser.tab.o simple_lexer.o Expr.o
	$(CC) $(CFLAGS) –o $@ $^

simple_parser.tab.h simple_parser.tab.c: simple_parser.y
	bison –d $<

simple_lexer.o: simple_parser.tab.h Expr.h
simple_parser.tab.o: Expr.h
```

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”:

```
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:

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