Compilers
Parse Generators and ASTs

LEIC

FEUP-FCUP

2022

This lecture

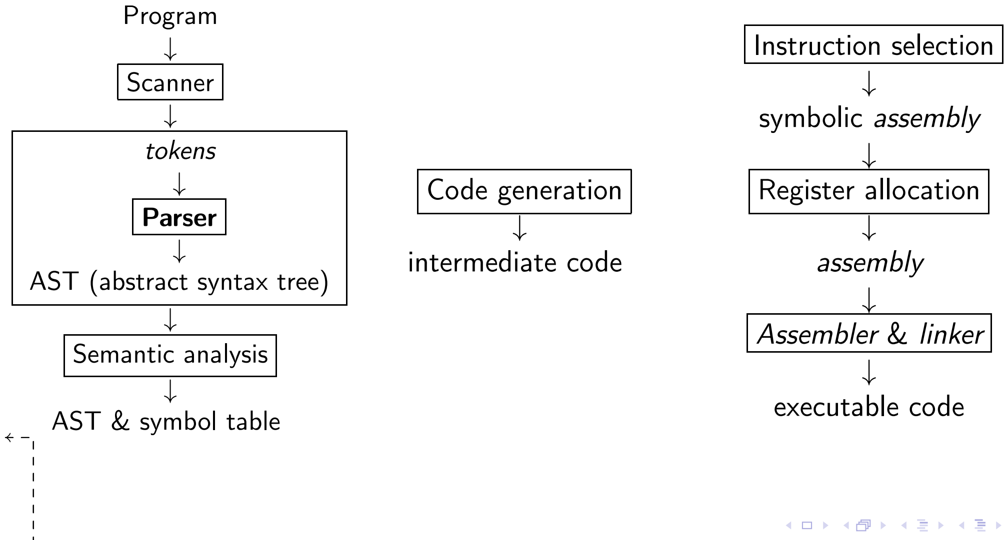
Parser generators

- *JavaCC*

Abstract Syntax

- Abstract Syntax
- Functional Languages
- C
- Java and OO language
- Other languages

Compiler


Program
Scanner
tokens
Parser
AST (abstract syntax tree)
Semantic analysis
AST & symbol table
Code generation
intermediate code
Instruction selection
symbolic *assembly*
Register allocation
assembly
Assembler & linker
executable code

This lecture

Parser generators

JavaCC

Abstract Syntax

Abstract Syntax

Functional Languages

C

Java and OO language

Other languages

Parser generators

- Parsers generators:

JavaCC	LL parser generator for Java
ANTLR	LL parser generator for Java
StableCC	LALR parser generator for Java, C++, C, Python and OCaml
Yacc	Unix LALR parser generator for C
Bison	a GNU version of Yacc for C and C++
Happy	LALR parser generator for Haskell

This lecture

Parser generators

- *JavaCC*

Abstract Syntax

- Abstract Syntax
- Functional Languages
- C
- Java and OO language
- Other languages

JavaCC

- LL Parser generator
- Grammar Rules are of the form:

```
void Assignment() : {} { Identifier() "=" Expression() ";" }
```

Example

```
PARSER_BEGIN(MyParser)
    public class MyParser {}
PARSER_END(MyParser)

SKIP :
{ " " | "\t" | "\n" }

TOKEN :
{ < WHILE: "while" >
| < BEGIN: "begin" >
| < END: "end" > | < DO: "do" > | < IF: "if" >
| < THEN: "then" >
| < ELSE: "else" > | < SEMI: ";" >
| < ASSIGN: "=" >
| < ID: ["a"-"z"](["a"-"z"] | ["0"-"9"])* > }
```

Example

```
void Prog() :
{}
{ StmList() <EOF> }

void StmList() :
{}
{ Stm() StmListPrime() }

void StmListPrime() :
{}
{ ( ";" Stm() StmListPrime() )? }
```

Example

```
void Stm() :
{}
{ <ID> "=" <ID>
| "while" <ID> "do" Stm()
| "begin" StmList() "end"
| LOOKAHEAD(5) /* we need to lookahead until the else */
  "if" <ID> "then" Stm()
| "if" <ID> "then" Stm() "else" Stm()
}
```

This lecture

Parser generators

JavaCC

Abstract Syntax

Abstract Syntax

Functional Languages

C

Java and OO language

Other languages

This lecture

Abstract Syntax

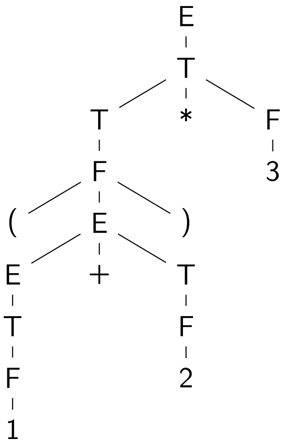
Consider the following grammar:

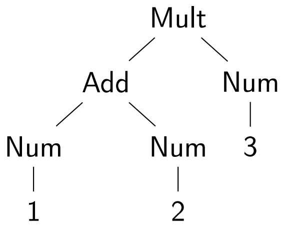
$$
\begin{array}{lll}
E \rightarrow E + T & T \rightarrow T * F & F \rightarrow \mathsf{num} \\
E \rightarrow T & T \rightarrow F & F \rightarrow (\ E\ )
\end{array}
$$

and the derivation:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow \cdots \Rightarrow \texttt{(1+2)*3}$$

Abstract Syntax (cont.)





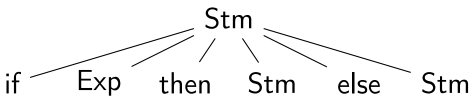
The parse tree on the left has redundant information.
The abstract syntax tree (AST) on the right has one node per operation.

Abstract Syntax (cont.)

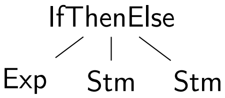
Example: commands.

$$\begin{aligned} &\mathit{Stm} \rightarrow \text{if } \mathit{Exp} \text{ then } \mathit{Stm} \text{ else } \mathit{Stm} \\ &\mathit{Stm} \rightarrow \mathit{etc.} \end{aligned}$$

In the parse tree each keyword is a different node.



In the AST we only need one node for the if/then/else:



This lecture

Functional Languages

In ML, OCaml, Haskell or F# the AST is an algebraic type:

- Alternatives identified by the constructors
- Datatypes may be recursive
- Processing AST uses pattern matching

Example

```
data Stm = AssignStm String Exp   -- ident = exp
         | IncrStm String         -- ident++
         | CompoundStm Stm Stm    -- stm1; stm2

data Exp = IdExp String           -- x, y, z, etc.
         | NumExp Int             -- 123, etc.
         | OpExp Exp BinOp Exp    -- e1+e2, e1*e2, ...

data BinOp = Plus | Minus | Times | Div
```

Example (cont.)

Example of an AST:

```
example :: Stm
example =
  CompoundStm
  (AssignStm "a"
    (OpExp (NumExp 5) Plus (NumExp 3))
  )
  (AssignStm "b"
    (OpExp (IdExp "a") Minus (NumExp 2))
  )
```

Example (cont.)

Processing using pattern matching:

```
process :: Stm -> ...
process (IncrStm id) = ...
process (AssignStm id exp) = ...
process (CompoundStm s1 s2) = ...
```

Or cases:

```
process stm = case stm of
   IncrStm id -> ...
   AssignStm id exp -> ...
   CompoundStm s1 s2 -> ...
```

This lecture

C

- A struct for each syntactic element:
  - a (*tag*) for the kind;
  - a union of alternatives;
- Constructors
- Use the *tag* to process the alternatives

Example

```
struct _stm {
  enum
   {COMPOUND, ASSIGN, INCR} tag;
  union {
    struct {      // for COMPOUND
      struct _stm *fst, *snd;
    } compound;
    struct {        // for ASSIGN
      char *ident;
      struct _exp *expr;
    } assign;
    char *ident;    // for INCR
  };
};

typedef struct _stm *Stm;
```

```
typedef enum {PLUS, MINUS, TIMES, DIV} binop;
struct _exp {
  enum {ID, NUM, OP} tag;
  union {
    int val;             // for NUM
    char *id;            // for ID
    struct {             // for OP
      binop op;
      struct _exp *left, *right;
    } binop;
  };
};
typedef struct _exp *Exp;
```

Example (cont.)

```
Exp mk_num(int v) {
  Exp e = (Exp) malloc(sizeof(struct _exp));
  e->tag = NUM;
  e->val = v;
  return e;
}

Exp mk_ident(char *txt) {
  Exp e = (Exp)malloc(sizeof(struct _exp));
  char *str = malloc(strlen(txt)+1);
  strcpy(str, txt);
  e->tag = ID;
  e->id = str;
  return e;
}
```

Example (cont.)

```
Exp mk_op(binop op, Exp e1, Exp e2) {
  Exp e = (Exp) malloc(sizeof(struct _exp));
  e->tag = OP;
  e->binop.op = op;
  e->binop.left = e1;
  e->binop.right = e2;
  return e;
}
```

(The other constructors are similar)

Example (cont.)

Example of an AST:

```
Stm example =
    mk_compound
    (mk_assign("a", mk_op(PLUS, mk_num(5), mk_num(3))),
     mk_assign("b", mk_op(TIMES, mk_ident("a"), mk_num(2))));
```

Example (cont.)

Building the AST in *Yacc/Bison*:

```
exp : term                { $$ = $1; }
    | exp '+' term        { $$ = mk_binop(PLUS, $1, $3); }
    ;

term : TOK_NUM            { $$ = mk_num($1); }
     |  '(' exp ')'       { $$ = $2; }
     ;
```

Example (cont.)

Processing the AST using tags:

```
void process(Stm stm) {
  switch(stm->tag) {
    case COMPOUND: // use stm->compound.fst and stm->compound.snd
      ⋮
    break;
    case ASSIGN:   // use stm->assign.ident and stm->assign.expr
      ⋮
    break;
    case INCR: // etc.
      ⋮
    break;
  }
}
```

This lecture

Parser generators

JavaCC

Abstract Syntax

Abstract Syntax

Functional Languages

C

Java and OO language

Other languages

Java

Em OO languages:

- An abstract class for each syntactic category;
- A subclass for each alternative
- To identify the alternatives use `instanceof` and *downcasts*

Example

```
public abstract class Stm {}

public class CompoundStm extends Stm
{
  public Stm fst, snd;
  public CompoundStm(Stm s1, Stm s2)
  {
     fst = s1; snd = s2;
  }
}
```

```
public class AssignStm extends Stm
{
  public String id;
  public Exp exp;
  public AssignStm(String i, Exp e)
  {
    id=i; exp=e;
  }
}
```

(Other subclasses for the other alternatives)

Example (cont.)

```
public abstract class Exp {}

public class IdExp extends Exp {
  public String id;
  public IdExp(String i) {
    id=i;
  }
}
```

```
public class NumExp extends Exp {
  public int num;
  public NumExp(int n) {
    num=n;
  }
}
```

```
public class PlusExp extends Exp {
  private Exp left, right;
  public PlusExp(Exp e1, Exp e2) {
    left=e1; right=e2;
  }
}
```

Example of an AST:

Example (cont.)

```
Stm prog =
   new CompoundStm(new AssignStm("a",
                                new PlusExp(new NumExp(5),
                                            new NumExp(3))),
                   new AssignStm("b",
                                new TimesExp (new IdExp("a"),
                                              new NumExp(2))));
```

Example (cont.)

Building the AST in *JavaCC*:

```
Exp Exp() :
{ Exp e1,e2; }
{ e1=Term()
    ( "+" e2=Term() { e1=new PlusExp(e1,e2); }
    | "-" e2=Term() { e1=new MinusExp(e1,e2); } )*
  { return e1; }
}
Exp Term() :
{ Exp e1,e2; }
{ e1=Factor()
    ( "*" e2=Factor() { e1=new TimesExp(e1,e2); }
    | "/" e2=Factor() { e1=new DivideExp(e1,e2); })*
  { return e1; }
}
```

Example (cont.)

```
Exp Factor() :
{ Token t; Exp e; }
  { ( t=<IDENTIFIER>        { return new IdExp(t.image); }  |
      t=<INTEGER_LITERAL>   { return new NumExp(t.image); } |
      "(" e=Exp() ")"       { return e; } )
 }
```

Example (cont.)

Processing using cases:

```
public void process(Stm stm) {
  if(stm instanceof CompoundStm) {
      CompoundStm cstm = (CompoundStm)stm;
      ⋮
  }
  else if(stm instanceof AssignStm) {
       AssignStm astm = (AssignStm)stm;
       ⋮
  }
  else ...
}
```

An interpreter

```
public abstract class Exp {public abstract int eval();}

public class IdExp extends Exp {
  public String id;
  public IdExp(String i) {
    id=i;
  }
  public int eval() {
       return lookup(id);
   }
}

public class NumExp extends Exp {
  public int num;
  public NumExp(int n) {
    num=n;
  }
  public int eval() {return num;}
}
```

An interpreter (cont.)

```
public class PlusExp extends Exp {
  private Exp left, right;
  public PlusExp(Exp e1, Exp e2) {
    left=e1; right=e2;
  }
  public int eval() {
       return left.eval()+right.eval();
   }
}
```

Visitors

```
public abstract class Exp {
      public abstract int accept(Visitor v);}
```

```
public class IdExp extends Exp {
  public String id;
  public IdExp(String i) {
    id=i;
  }
  public int accept(Visitor v) {
         return v.visit(this);
     }
}
```

```
public class NumExp extends Exp {
  public int num;
  public NumExp(int n) {
    num=n;
  }
  public int accept() {
         return v.visit(this);
     }
}
```

Visitors (cont.)

```
public class PlusExp extends Exp {
  private Exp left, right;
  public PlusExp(Exp e1, Exp e2) {
    left=e1; right=e2;
  }
  public int accept(Visitor v) {
         return v.visit(this);
     }
  }
```

Visitors (cont.)

```
public interface Visitor {
    public int visit(PlusExp e);
    public int visit(IdExp e);
    public int visit(NumExp e);
    .........
}
```

Visitors (cont.)

```
public class Interpreter implements Visitor {
    public int visit(PlusExp e) {
       return e.left.accept(this) + e.right.accept(this);
    }
    public int visit(IdExp e) {
       return lookup(e.id);
    }
    public int visit(NumExp e) {
       return e.num;
    }
```

This lecture

Parser generators

JavaCC

Abstract Syntax

Abstract Syntax

Functional Languages

C

Java and OO language

Other languages

Other languages

- Modern multi-paradigm languages (OO+Funcional) use pattern matching (as in Haskell):
  - “case classes” em Scala
  - “enumerations” em Swift e Rust