Sequence 2.4 – Syntax Analysis

P. de Oliveira Castro S. Tardieu

Syntax Analysis

A parser transforms a flow of tokens into an Abstract Syntax Tree

```
let var a := 10 in print_int(a+1) end
```

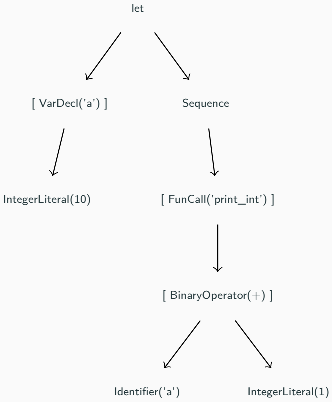

Figure 1: Translation into AST

Bison (Yacc) rules

To generate the parser we use Bison, which is a parser generator. From a set of *grammar rules* and *production rules* it automatically generates a program that generates an AST.

- *terminals* are Lexer Token (ID, INT, VAR)
- *non-terminals* correspond to a token produced by a production rule.
- A *grammar rule* is of the form, $\alpha \to \beta_1\beta_2\ldots\beta_k$ with α non-terminal and β_i either terminal or non-terminal. It triggers a production rule that produces a non-terminal of type α.
- The $\to$ is also written $:$ in Bison and means that we can encode the right hand side (RHS) with an AST node of type α.

Exemple of Bison Rules

```
expr : INT
  { $$ = new IntegerLiteral(@1, $1); };
```

The first line is the grammar rule. It means that if we find a lexer INT token we can produce a non-terminal `expr`.

The second line is the production rule. It tells Bison how to build the Expression node.

- `$$` is the result of the production rule
- `@1` is the source location of β_1, here INT
- `$1` is the value of β_1, here INT

Here, we show Bison that to produce an expression from an INT token he need to produce an `IntegerLiteral` node.

This rule transforms the source code `101` into the AST node `IntegerLiteral(101)`.

A more complex example

Tiger source code example:

```
var a : int := 10
```

```
varDecl : VAR ID typeannotation ASSIGN expr
  { $$ = new VarDecl(@1, $2, $5, $3); };
```

- $$ is the result of the production rule
- @1 is the source location of VAR (β_1)
- $2 is the value of ID ($\beta_2$)
- $5 is the result of the production rule for expr (β_5)
- $3 is the result of the production rule for typeannotation (β_3)

A more complex example

```
varDecl : VAR ID typeannotation ASSIGN expr
  { $$ = new VarDecl(@1, $2, $5, $3); };
```

```
var a : int := 10
```

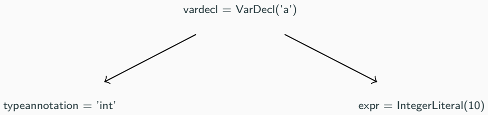

Figure 2: Translation into AST

The type returned by a production rule must be declared in the preamble with,

```
%type <VarDecl *> varDecl
```

Disjunctive rules

- Sometimes a non-terminal can capture different RHS
- There are two equivalent ways to express this,

```
expr: INT { $$ = new IntegerLiteral(@1, $1); }
    | ID  { $$ = new Identifier(@1, $1); }
;

expr: INT { $$ = new IntegerLiteral(@1, $1); }
;
expr: ID  { $$ = new Identifier(@1, $1); }
;
```

Recursive rules

```
1 ; 2 ; 3; 4 ; 5
```

Bison

```
%type <std::vector<Expr *>> nonemptyexprs;
...

/* List is composed of a single expression */
nonemptyexprs : expr { $$ = std::vector<Expr*>({$1}); } ;
```

```
 1 ; 2 ; 3 ; 4  ;   5
\_____________/   \___/
 nonemptyexprs     expr
```

```
/* List is composed of a multiple expressions */
nonemptyexprs : nonemptyexprs SEMICOLON expr /* a recursive rule */
  { $$ = std::move($1); /* $1 is not used anymore */
    $$.push_back($3); };
```

Parser conflicts

- Grammar rules can be ambiguous,

```
expr : expr PLUS expr
     | expr TIMES expr;
```

```
4 + 2 * 3
```

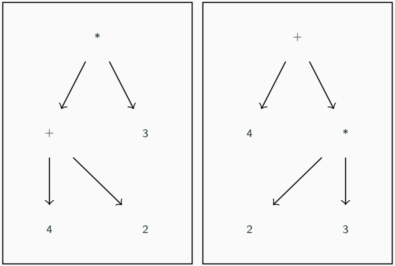

Figure 3: Translation into AST

Precedence Rules

- To force the parser to chose the right version we use precedence rules,
  - + − * / are *left-associative*
  - + and − are less binding than * and /

Bison

```
%left PLUS MINUS;
%left TIMES DIVIDE;
```

How does the parser works?

- Bison works by generating the AST bottom-up.
  - Whenever it matches the RHS of a rule, it replaces it with a non-terminal. . .
  - . . . the non-terminal is a sub-tree . . .
  - . . . which in turn may appear in the RHS of a later rule.
- It is not always wise to apply a rule as soon as possible (see previous slide)
- To decide when to apply a rule, Bison uses Stack Automatas which are more complex versions of the DFAs we saw in last sequence.