ساخت یک مفسر با flex و bison

پارسا نیاوند

Flex و Bison: ابزارهای ساخت کامپایلر

Flex (تولیدکننده تحلیلگر لغوی - Lexer)

- **فایل ورودی:** `lang.l`
- **وظیفه:** اسکن کردن کد منبع و شکستن آن به واحدهای معنادار به نام **توکن (Token)**.
  - `let` ← `LET_KEYWORD`
  - `123` ← `INTEGER`
  - `my_var` ← `IDENTIFIER`
- **مثال:** این فرآیند مانند شکستن یک جمله فارسی به کلمات و علائم نگارشی مجزا است.

Flex و Bison: ابزارهای ساخت کامپایلر

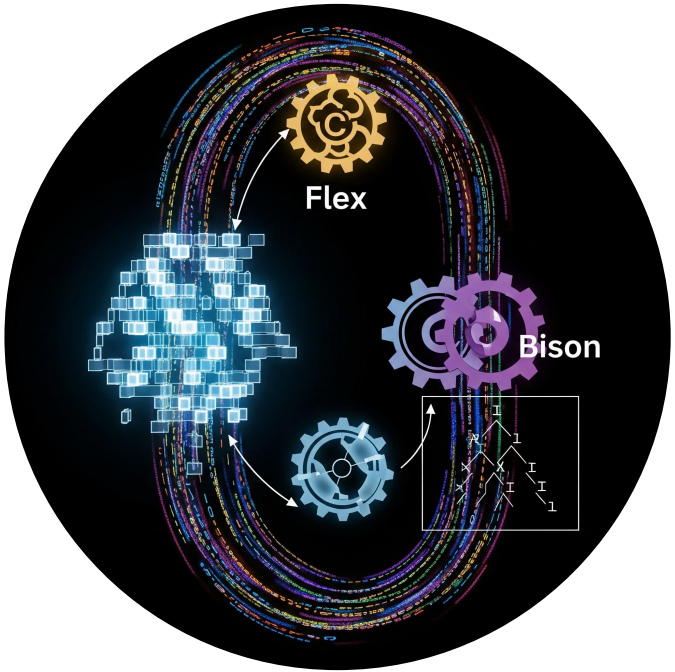
Bison (تولیدکننده تجزیه‌گر - Parser)

- **فایل ورودی:** `lang.y`
- **وظیفه:** دریافت توکن‌ها از Flex و بررسی اینکه آیا ساختار گرامری آن‌ها صحیح است یا خیر. با شناسایی ساختارهای معتبر (مثل تعریف متغیر)، کدهای C مربوطه را برای اجرای عملیات اجرا می‌کند.
- **مثال:** این فرآیند مانند بررسی این است که آیا دنباله‌ای از کلمات در یک جمله فارسی، ساختار گرامری درستی (مانند نهاد-فعل-مفعول) را تشکیل می‌دهند یا خیر.

مفسر زبان: توضیحات و اجزا

01 ——— برای ساخت این مفسر از flex و bison استفاده میکنیم

02 ——— این مفسر الهام گرفته شده از زبان C میباشد.



03 ——— توکن ها با استفاده از flex از متن استخراج میشوند.

04 ——— سپس bison توکن ها را گرفته و با استفاده از گرامر آن را اجرا میکند.

ویژگی‌های زبان

این زبان برنامه‌نویسی ساده، قابلیت‌های اصلی زیر را پشتیبانی می‌کند:

- **نقطه ورود (Entry Point):** تمام کدها باید داخل یک بلوک اصلی `int main() { ... }` قرار گیرند. این تابع، نقطه شروع اجرای برنامه است.
- **تعریف متغیر (Variable Definition):** می‌توانید متغیرهای صحیح را با استفاده از کلمه کلیدی **`let`** تعریف و مقداردهی اولیه کنید.
  - مثال: `let x = 10;`
- **چاپ در کنسول (Console Output):** برای نمایش مقدار یک متغیر یا نتیجه یک عبارت در خروجی، از تابع **`show()`** استفاده می‌شود.
  - مثال: `show(x * 2);`

- **ورودی کاربر:** جهت دریافت یک عدد صحیح از کاربر، می‌توانید از تابع `vorood()` استفاده کنید.
  - مثال: `let age = vorood();`

- **عبارات حسابی:** این زبان از چهار عمل اصلی حسابی **+** (جمع)، **-** (تفریق)، ***** (ضرب) و **/** (تقسیم) پشتیبانی می‌کند. حق تقدم استاندارد عملگرها (تقدم ضرب و تقسیم بر جمع و تفریق) رعایت می‌شود.

- **توضیحات:** برای نوشتن توضیحات تک‌خطی که توسط کامپایلر نادیده گرفته می‌شوند، از **//** استفاده کنید.
  - مثال: `This is a comment //`

برنامه نمونه

```
1  int main() {
2      show(11);
3      let a = 10;
4      let b = 20;
5      let c = a + b;
6      show(c); // shows 30
7      let d = a+b+c*2;
8      show(d);
9
10     let k = vorood();
11     show(k*2+1);
12 }
```

خط 2-1: برنامه با `int main()` آغاز شده و ابتدا عدد `11` را در خروجی نمایش می‌دهد.

خط 6-5: متغیر `c` تعریف شده و حاصل جمع `a` و `b` (که برابر `30` است) در آن ذخیره و سپس نمایش داده می‌شود.

خط 8-7: متغیر `d` با استفاده از یک عبارت حسابی پیچیده‌تر مقداردهی می‌شود.

خط 11-10: متغیر `k` تعریف شده و منتظر دریافت یک عدد صحیح از کاربر (`vorood()`) می‌ماند. سپس حاصل `k * 2 + 1` را محاسبه و در خروجی نمایش می‌دهد. (برای مثال اگر کاربر `5` را وارد کند، `11` نمایش داده می‌شود).

فایل lang.l - تحلیلگر لغوی (Lexer)

این فایل با استفاده از **عبارات منظم (Regular Expressions)**، «کلمات» یا همان **توکن‌های** زبان ما را تعریف می‌کند.

ابزار **Flex** این فایل را می‌خواند تا یک تابع به زبان C به نام `yylex()` را تولید کند. این تابع وظیفه دارد کد منبع را کاراکتر به کاراکتر بخواند و در هر بار فراخوانی، یک توکن معنادار را برگرداند.

فایل lang.l از سه بخش تشکیل شده است:

- بخش تعاریف: شامل کد ها و تعاریف C
- بخش قوانین: الگو و عبارات منظم به توکن ربط داده میشوند
- بخش کد کاربر: شامل کد های کمکی C که به فایل خروجی اضافه میشوند

بخش اول: تعاریف

این بخش شامل کدهای C، گزینه‌های Flex و نام‌گذاری عبارات منظم است که قبل از شروع قوانین تحلیل لغوی تعریف می‌شوند.

```
%{
#include "lang.tab.h" // این فایل توسط Bison ساخته می‌شود
#include <string.h>
#include <stdlib.h>
%}

%option noyywrap
%option yylineno

DIGIT    [0-9]
LETTER   [a-zA-Z_]
ID       {LETTER}({LETTER}|{DIGIT})*
```

بلوک %{ ... %}

- کدهای داخل این بلوک مستقیماً به ابتدای فایل C خروجی کپی می‌شوند.
- **"lang.tab.h":** تعاریف عددی توکن‌ها (مانند INT_KEYWORD, IDENTIFIER) با این کار، تحلیلگر لغوی (Flex) با تحلیلگر گرامری (Bison) هماهنگ می‌شود.
- دو خط دیگر برای تعریف متغیر و دستکاری رشته ها می باشد.

```
%{
#include "lang.tab.h"
#include <string.h>
#include <stdlib.h>

%}
```

نام‌گذاری عبارات منظم و گزینه

گزینه‌های %option:

- **%option noyywrap** : یک گزینه استاندارد که کامپایل را برای یک فایل ورودی ساده می‌کند.
- **%option yylineno** : به Flex دستور می‌دهد تا شماره خط فعلی را به صورت خودکار در متغیر yylineno ذخیره کند (برای پیدا کردن ارور)

```
8  %option noyywrap
9  %option yylineno
10
11 DIGIT    [0-9]
12 LETTER   [a-zA-Z_]
13 ID       {LETTER}({LETTER}|{DIGIT})*
```

نام‌گذاری عبارات منظم:

- تعاریف الگو های زبان برای خوانایی کد
- **DIGIT**: ارقام
- **LETTER**: حروف الفبا (کوچک یا بزرگ) یا _.
- **ID**: تعریف الگوی یک شناسه (Identifier):

باید با حرف الفبا شروع شود و بعد آن حروف یا عدد (DIGIT یا LETTER) بیاید.

بخش دوم: قوانین

در این بخش الگوهای متنی (عبارات منظم) را به توکن‌های مشخصی که برای تجزیه‌گر (Bison) ارسال می‌شوند، مرتبط می‌کنیم.

```
"int"         { return INT_KEYWORD; }
"main"        { return MAIN_KEYWORD; }
"let"         { return LET_KEYWORD; }
"show"        { return SHOW_KEYWORD; }
"vorood"      { return INPUT_KEYWORD; }

{DIGIT}+      { yylval.ival = atoi(yytext); return INTEGER; }
{ID}          { yylval.sval = strdup(yytext); return IDENTIFIER; }
```

قوانین

- **تعریف تابع اصلی**
  - **الگو:** `"int", "main"`
  - **توکن:** `INT_KEYWORD, MAIN_KEYWORD`
  - **توضیح:** این کلمات برای تعریف نقطه شروع برنامه (`int main`) استفاده می‌شوند تا ساختار آن شبیه به زبان C++ باشد.

- **تعریف متغیر**
  - **الگو:** `"let"`
  - **توکن:** `LET_KEYWORD`
  - **توضیح:** هرگاه تحلیلگر لغوی کلمه `let` را ببیند، به تجزیه‌گر اطلاع می‌دهد که تعریف یک متغیر در حال شروع است.

قوانین

- **چاپ در خروجی**
  - **الگو:** `"show"`
  - **توکن:** `SHOW_KEYWORD`
  - **توضیح:** این کلمه کلیدی برای فراخوانی تابع چاپ مقادیر در کنسول به کار می‌رود.

- **دریافت ورودی**
  - **الگو:** `"vorood"`
  - **توکن:** `INPUT_KEYWORD`
  - **توضیح:** این کلمه کلیدی برای فراخوانی تابع دریافت ورودی از کاربر استفاده می‌شود.

قوانین

- **اعداد صحیح (Integers)**
  - **الگو:** `{DIGIT}+` (یک یا چند رقم پشت سر هم)
  - **عملیات:** `yylval.ival = atoi(yytext); return INTEGER;`
  - **توضیح:**

متن تطابق‌یافته (مثلاً "123") که در متغیر **`yytext`** قرار دارد، به عدد صحیح تبدیل می‌شود.

این مقدار عددی در متغیر مشترک با Bison به نام **`yylval`** ذخیره می‌گردد.

نوع توکن `INTEGER` به تجزیه‌گر برگردانده می‌شود.

قوانین

- **شناسه‌ها (Identifiers)**
  - **الگو:** `{ID}` (نام یک متغیر مانند "my_var")
  - **عملیات:** `yylval.sval = strdup(yytext); return IDENTIFIER;`
  - **توضیح:**

1. یک **کپی** از رشته‌ی شناسه (که در `yytext` است) در حافظه ایجاد شده و آدرس آن در `yylval` ذخیره می‌شود. (استفاده از `strdup` ضروری است چون `yytext` در تطابق بعدی بازنویسی می‌شود).
2. نوع توکن `IDENTIFIER` به تجزیه‌گر برگردانده می‌شود.

قوانین

- **نادیده گرفتن کاراکترها**
  - **الگوها:** `[ \t\n]+` (فضاهای خالی) و `"//".*` (کامنت)
  - **عملیات:** `{}` (هیچ عملیاتی)
  - **توضیح:** تحلیلگر لغوی این الگوها را شناسایی کرده و به سادگی نادیده می‌گیرد، بنابراین هرگز به دست تجزیه‌گر نمی‌رسند.

- **مدیریت خطا**
  - **الگو:** `.` (هر کاراکتر تکی دیگر)
  - **عملیات:** `fprintf("...");` (چاپ خطا)
  - **توضیح:** این قانون در انتها قرار دارد و هر کاراکتری را که با هیچ‌یک از قوانین بالا تطابق نداشته باشد، به عنوان یک کاراکتر غیرمجاز شناسایی کرده و خطا می‌دهد.

قوانین

- **عملیات ریاضی :** کاراکترهای +, -, *, / به ترتیب به توکن‌های ADD_OP, SUB_OP, MUL_OP و DIV_OP تبدیل می‌شوند تا تجزیه‌گر بتواند عبارات حسابی را ارزیابی کند.

- **پرانتزها و جداکننده‌ها :** نمادهای ( ) برای گروه‌بندی عبارات و { } برای تعریف بلوک‌های کد استفاده شده و به توکن‌های LPAREN/RPAREN و LBRACE/RBRACE نگاشت داده می‌شوند.

- **پایان دستورات :** کاراکتر سمی‌کولن (;) به عنوان یک توکن حیاتی (SEMICOLON) به تجزیه‌گر اعلام می‌کند که یک دستورالعمل کامل به پایان رسیده است.

```
27  "+"         { return ADD_OP; }
28  "-"         { return SUB_OP; }
29  "*"         { return MUL_OP; }
30  "/"         { return DIV_OP; }
31  "="         { return ASSIGN_OP; }
32
33  "("         { return LPAREN; }
34  ")"         { return RPAREN; }
35  "{"         { return LBRACE; }
36  "}"         { return RBRACE; }
37  ";"         { return SEMICOLON; }
```

فایل lang.y - تجزیه‌گر و مفسر

دو وظیفه اصلی این فایل:

1. **تعریف گرامر (Grammar):** مشخص می‌کند که چه ترکیبی از توکن‌ها یک برنامه معتبر را تشکیل می‌دهند (مانند `let IDENTIFIER = expression;`).
2. **تعریف مفسر (Interpreter):** شامل کدهای C است که با شناسایی هر بخش معتبر از گرامر، عملیات مربوط به آن را اجرا می‌کنند.

ابزار **Bison** از این فایل برای تولید یک تابع C به نام `yyparse()` استفاده می‌کند که جریان توکن‌ها را از تحلیلگر لغوی دریافت و پردازش می‌کند.

بخش اول: مقدمه C و تعاریف Bison

قبل از تعریف قوانین گرامری، ابتدا باید ابزارها و داده‌های مورد نیاز خود را در بخش مقدماتی فایل `lang.y` آماده کنیم.

این بخش به دو قسمت اصلی تقسیم می‌شود:

1. **بلوک کد C (`%{ ... %}`):** شامل توابع کمکی و ساختارهای داده‌ای است که به عنوان حافظه و منطق مفسر عمل می‌کنند.
2. **تعاریف `%token`, `union`:** توکن‌ها، انواع داده و قوانین حق تقدم عملگرها را به Bison معرفی می‌کند.

بلوک کد C: حافظه مفسر - جدول نمادها

برای اینکه مفسر بتواند متغیرها را به خاطر بسپارد، از یک **جدول نمادها (Symbol Table)** استفاده می‌کنیم.

```c
#define MAX_SYMBOLS 100
typedef struct {
    char *name;
    int value;
} Symbol;

Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

// symbol table vars val
int lookupSymbol(char *name) {···
}

//storing vars and values
void storeSymbol(char *name, int value) {···
}
```

- ظرفیت ثابت می‌تواند حداکثر ۱۰۰ متغیر را در حافظه نگهداری کند (`MAX_SYMBOLS`). یک شمارنده به نام `symbolCount` تعداد متغیرهایی که در لحظه تعریف شده‌اند را پیگیری می‌کند.
- هر متغیر به صورت یک رکورد `Symbol` ذخیره می‌شود که شامل دو بخش است: نام متغیر (`name`) و مقدار عددی آن (`value`). تمام این رکوردها در یک آرایه به نام `symbolTable` نگهداری می‌شوند که حافظه اصلی برنامه را تشکیل می‌دهد.

بلوک کد C: حافظه مفسر - جدول نمادها

- **`lookupSymbol`**: مقدار یک متغیر را با استفاده از نام آن از جدول بازیابی می‌کند.

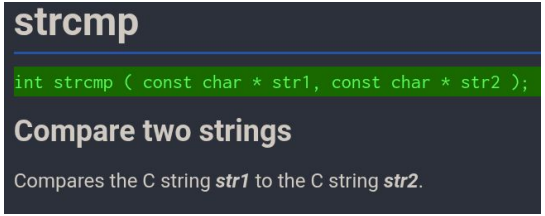
این تابع نام یک متغیر را به عنوان ورودی دریافت کرده و در نماد ها به دنبال آن می‌گردد, اگر متغیر پیدا شود، مقدار آن را برمی‌گرداند. اما اگر پس از جستجوی کامل، متغیری با آن نام یافت نشود، به این معناست که متغیر تعریف نشده است

```c
// symbol table vars val
int lookupSymbol(char *name) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return symbolTable[i].value;
        }
    }
    fprintf(stderr, "Line %d: Undefined variable '%s'\n", yylineno, name);
    exit(EXIT_FAILURE);
}
```

بلوک کد C: حافظه مفسر - جدول نمادها

- **`storeSymbol`**: یک متغیر جدید را به جدول اضافه می‌کند یا مقدار یک متغیر موجود را به‌روزرسانی می‌کند.

```c
//storing vars and values
void storeSymbol(char *name, int value) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            symbolTable[i].value = value;
            return;
        }
    }
    if (symbolCount < MAX_SYMBOLS) {
        symbolTable[symbolCount].name = name;
        symbolTable[symbolCount].value = value;
        symbolCount++;
    } else {
        fprintf(stderr, "Line %d: Symbol table overflow\n", yylineno);
        exit(EXIT_FAILURE);
    }
}
```



تعاریف Bison: %union - تعریف انواع داده

```
%union {
    int ival;
    char *sval;
}
```

این دستور یک union در زبان C تعریف می‌کند که تمام انواع داده‌ای ممکن برای مقادیر گرامری را در خود جای می‌دهد.

هدف این است که به Bison بگویید چه انواع داده‌ای ممکن است بین تحلیلگر لغوی (Flex) و تجزیه‌گر (Bison) جابجا شوند. Bison از این اطلاعات استفاده می‌کند تا **به صورت خودکار یک union استاندارد C** (معمولاً با نام YYSTYPE) در فایل خروجی .c خود تولید کند.

```
/* Value type.  */
#if ! defined YYSTYPE && ! defined Y
union YYSTYPE
{
#line 104 "lang.y"

    int ival;
    char *sval;

#line 88 "lang.tab.h"
```

تعاریف Bison: %token - معرفی توکن‌ها

این بخش شامل گزینه‌های Flex و نام‌گذاری عبارات منظم است که قبل از شروع قوانین تحلیل لغوی تعریف می‌شوند.

- **%token <ival> INTEGER**: به Bison می‌گوید:
  1. توکنی به نام `INTEGER` وجود دارد, مقدار مرتبط با آن از نوع عدد صحیح است و باید در فیلد `ival` از `yylval` ذخیره شود.

- **%token <sval>**: توکن `IDENTIFIER` یک مقدار رشته‌ای دارد که در فیلد `sval` ذخیره می‌شود.
- توکن‌هایی مانند `LET_KEYWORD` که مقدار خاصی حمل نمی‌کنند، نیازی به تعریف نوع ندارند.

```
%token <ival> INTEGER
%token <sval> IDENTIFIER
%token INT_KEYWORD MAIN_KEYWORD LET_KEYWORD SHOW_KEYWORD
INPUT_KEYWORD
%token LPAREN RPAREN LBRACE RBRACE SEMICOLON ASSIGN_OP
%token ADD_OP SUB_OP MUL_OP DIV_OP

%type <ival> expression term factor primary
```

Let x = 10 ;
(LET_KEYWORD IDENTIFIER ASSIGN_OP INTEGER SEMICOLON)

تعاریف Bison: %left - حق تقدم عملگرها

این دستورات **شرکت‌پذیری (Associativity)** و **حق تقدم (Precedence)** عملگرها را تعریف می‌کنند.

- `%left` یعنی عملگرها شرکت‌پذیری از چپ به راست دارند
- **قانون حق تقدم:** خطوطی که **زودتر** تعریف می‌شوند، حق تقدم **کمتری** دارند.
- در اینجا، `ADD_OP` و `SUB_OP` حق تقدم کمتری نسبت به `MUL_OP` و `DIV_OP` دارند، که دقیقاً همان چیزی است که برای محاسبات ریاضی استاندارد نیاز داریم.

```
%left ADD_OP SUB_OP
%left MUL_OP DIV_OP
```

تعاریف Bison: %type - تعیین نوع قوانین

همانطور که برای توکن‌ها نوع داده تعریف کردیم، برای نتایج حاصل از قوانین گرامری (نمادهای غیرپایانی) نیز باید نوع داده مشخص کنیم.

- `%type <ival> expression` : این دستور به Bison می‌گوید:
  - یک قانون در گرامر به نام `expression` وجود دارد.
  - نتیجه نهایی حاصل از ارزیابی این قانون، یک مقدار عددی صحیح است.
  - این نتیجه باید در فیلد `ival` ذخیره شود تا در قوانین سطح بالاتر قابل استفاده باشد.

نتیجه نهایی حاصل از ارزیابی چهار قانون گرامری اصلی ما (`expression`, `term`, `factor`, `primary`) همیشه یک **مقدار عددی صحیح** خواهد بود

```
%type <ival> expression term factor primary
```

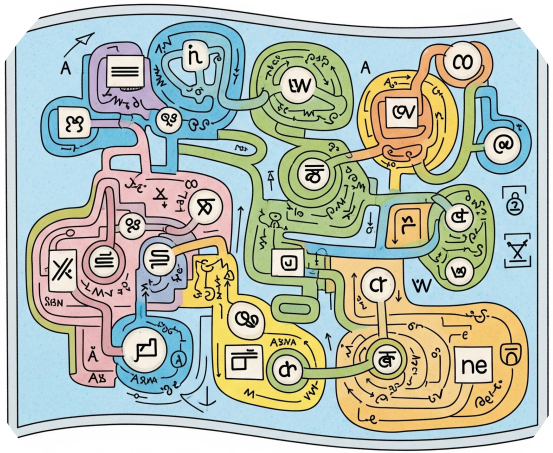
مثال:

- **Expression (عبارت):** عمومی‌ترین مفهوم است که یک یا چند `term` را نمایندگی می‌کند و به کل محاسبه در سمت راست یک تخصیص گفته می‌شود.
- **Term (جمله):** گروهی از یک یا چند `factor` است که با عملگرهای جمع (`+`) یا تفریق (`-`) به هم متصل شده‌اند.
- **Factor (عامل):** گروهی از یک یا چند `primary` است که با عملگرهای ضرب (`*`) یا تقسیم (`/`) به هم متصل شده‌اند.
- **Primary (اولیه):** پایه‌ای‌ترین و غیرقابل‌تجزیه‌ترین واحد در یک عبارت است.

```
13    let result = 10 * (x + 5);
```

- `Primary: 10, x, 5,  (x + 5)`
- `Factor: 10, (x + 5), 10 * (x + 5)`
- `Term: x + 5, 10 * (x + 5)`
- `Expression: 10 * (x + 5)`

قوانین گرامر



- بخش قوانین گرامر، نقشه راه ساختاری زبان است
- با تعریف قوانین، به Bison گرامر معتبر را آموزش می‌دهیم
- این بخش، قلب تجزیه‌گر محسوب می‌شود
- دنباله‌ای از توکن‌ها، برنامه معتبر را تشکیل می‌دهد

- نام_قانون: جزء اول جزء دوم ... { کد C برای اجرا };

متغیرهای ویژه در قوانین Bison

Bison برای دسترسی به مقادیر توکن‌ها و قوانین، متغیرهای خاصی را در اختیار ما قرار می‌دهد:

- **$$ (مقدار خروجی):** نشان‌دهنده **مقدار نتیجه** قانون فعلی است. این متغیر مانند `return value` در یک تابع عمل می‌کند.
- **$1, $2, $3, ... (مقادیر ورودی):** به ترتیب به مقدار **اجزای سمت راست قانون** از چپ به راست اشاره دارند. این‌ها مانند پارامترهای ورودی یک تابع هستند.

```
expression: expression ADD_OP term { $$ = $1 + $3; } ($$) ($1) ($2) ($3)
```

قانون program: ساختار کلی برنامه

این قانون، بالاترین سطح گرامر و نقطه شروع تجزیه است. این قانون مشخص می‌کند که یک برنامه کامل و معتبر در زبان ما باید چه شکلی داشته باشد.

ترجمه: یک برنامه معتبر باید با `int main() { ... }` شروع شده و در داخل آکولادها، شامل دنباله‌ای از دستورات (`statements`) باشد.

عملکرد: کد C داخل `{}` تنها زمانی اجرا می‌شود که کل برنامه با موفقیت و بدون خطا تجزیه شود.

```
103  /* Grammar Rules */
104  program:
105      INT_KEYWORD MAIN_KEYWORD LPAREN RPAREN LBRACE
         statements RBRACE { printf("execution executed.\n"); }
106      ;
```

قانون statements: تعریف دنباله‌ای از دستورات

چگونه به Bison بگوییم که برنامه می‌تواند شامل صفر، یک یا بی‌نهایت دستور باشد؟ با استفاده از یک قانون بازگشتی (Recursive).

حالت اول (/* خالی */): یک لیست از دستورات می‌تواند خالی باشد (یعنی هیچ دستوری داخل `main` وجود نداشته باشد).

حالت دوم (`statements statement`): یا یک لیست از دستورات، از یک لیست کوچکتر (`statements`) و به دنبال آن یک دستور جدید (`statement`) تشکیل شده است. این الگو به Bison اجازه می‌دهد تا دستورات را یکی پس از دیگری بخواند.

```
108  statements:
109      /* empty */
110      | statements statement
111      ;
```

قانون statement: دستور تعریف متغیر (let)

این قانون، انواع دستورالعمل‌های معتبر در زبان ما را تعریف می‌کند. اولین نوع، تعریف و مقداردهی متغیر است.

- **الگو:** این قانون منتظر الگوی `let name = expression;` است.
- **مثال:** برای `let x = 10;`
  - `$2`: مقدار توکن `IDENTIFIER` خواهد بود (رشته "x").
  - `$4`: مقدار نهایی و محاسبه‌شده `expression` خواهد بود (عبارت: عدد 10).
- **عملکرد:** تابع `storeSymbol` فراخوانی شده و نام متغیر ("x") به همراه مقدار محاسبه‌شده (10) در جدول نمادها ذخیره می‌شود.

```
113 statement:
114     LET_KEYWORD IDENTIFIER ASSIGN_OP expression SEMICOLON {
115         storeSymbol($2, $4);
116     }
```

قانون statement: دستور چاپ (show)

دومین نوع دستور معتبر در زبان، چاپ کردن مقدار یک عبارت در خروجی است.

- **الگو:** این قانون منتظر الگوی نمایش عبارت است.
- **مثال:** برای `show(x * 2);`
  - `$3`: مقدار نهایی و محاسبه‌شده `expression` (یعنی مقدار `x` ضربدر ۲) خواهد بود.
- **عملکرد:** تابع `printf` فراخوانی شده و مقدار عددی محاسبه‌شده را در کنسول چاپ می‌کند.

```
113  statement:
114  ›   { ...
117      }
118      | SHOW_KEYWORD LPAREN expression RPAREN SEMICOLON {
119          printf("%d\n", $3);
```

مدیریت حق تقدم عملگرها

چگونه به تجزیه‌گر بفهمانیم که در عبارت `3 + 4 * 5`، ابتدا باید ضرب انجام شود؟

راه حل: با استفاده از یک ساختار گرامری لایه‌ای و سلسله‌مراتبی. ما به جای یک قانون کلی برای همه محاسبات، آن را به چهار سطح تقسیم می‌کنیم:

1. `expression` (کمترین اولویت: جمع و تفریق)
2. `term`
3. `factor`
4. `primary` (بیشترین اولویت: اعداد، متغیرها)

سطح ۱: قانون expression (جمع و تفریق)

این قانون در بالاترین سطح هرم محاسبات قرار دارد، بنابراین عملگرهای آن **آخرین** اولویت را دارند.

```
122  expression:
123      term                          { $$ = $1; }
124      | expression ADD_OP term      { $$ = $1 + $3; }
125      | expression SUB_OP term      { $$ = $1 - $3; }
126      ;
```

ترجمه: یک `expression` از یک یا چند `term` تشکیل شده که با عملگرهای `+` یا `-` به هم متصل شده‌اند.

عملکرد: وقتی Bison عبارتی مانند `a - b` را می‌بیند، ابتدا مقادیر `a` و `b` (که هر دو `term` هستند) را محاسبه کرده و سپس تفریق را انجام می‌دهد و نتیجه را در `$$` قرار می‌دهد.

سطح ۲: قانون term (ضرب و تقسیم)

این قانون در سطح میانی قرار دارد و اولویت آن بالاتر از جمع و تفریق است.

```
128 term:
129     factor                          { $$ = $1; }
130     | term MUL_OP factor            { $$ = $1 * $3; }
131     | term DIV_OP factor            {
132                                       if ($3 == 0) {
```

ترجمه: یک `term` از یک یا چند `factor` تشکیل شده که با عملگرهای `*` یا `/` به هم متصل شده‌اند.

عملکرد: این ساختار تضمین می‌کند که در عبارت `3 + 4 * 5`، تجزیه‌گر ابتدا مجبور است `4 * 5` را به عنوان یک `term` محاسبه کند تا بتواند به قانون `expression` برگردد.

سطوح ۳ و ۴: factor و primary

این دو قانون، پایه‌ای‌ترین بلوک‌های سازنده عبارات ریاضی هستند.

factor: در گرامر فعلی، یک پوشش ساده برای `primary` است (عامل).

primary: اتم یا کوچکترین جزء یک عبارت است که بیشترین اولویت را دارد.

```
140  factor:
141      primary                        { $$ = $1; }
142      ;
143
144  primary:
145      INTEGER                        { $$ = $1; }
146      | IDENTIFIER                   { $$ = lookupSymbol($1);
         free($1); }
147      | LPAREN expression RPAREN     { $$ = $2; }
148      | INPUT_KEYWORD LPAREN RPAREN { $$ = readIntegerInput
         (); }
149      ;
```

قانون `primary` مشخص می‌کند که چه چیزهایی می‌توانند به عنوان یک واحد پایه در محاسبات به کار روند:

1. **`INTEGER`**: یک عدد خام مانند `123`. مقدار آن مستقیماً استفاده می‌شود.
2. **`IDENTIFIER`**: یک متغیر مانند `x`. مقدار آن با `lookupSymbol` از جدول نمادها خوانده می‌شود.
3. **`LPAREN expression RPAREN`**: یک عبارت داخل پرانتز مانند `(2 + 5)`. پرانتزها اولویت محاسبات را به زور بالا می‌برند.
4. **`INPUT_KEYWORD LPAREN RPAREN`**: فراخوانی تابع `vorood()`. مقدار آن از ورودی کاربر خوانده می‌شود.

تابع main(): نقطه شروع و پایان برنامه

تابع `main` به عنوان نقطه ورود برنامه، وظایف ساده و مهمی را بر عهده دارد.

1. **فراخوانی `yyparse()`:** این تابع که توسط Bison ساخته شده، موتور اصلی مفسر را روشن کرده و فرآیند تجزیه کد منبع را آغاز می‌کند.
2. **مدیریت حافظه:** پس از پایان کار `yyparse`، حلقه `for` اجرا شده و تمام حافظه‌ای که برای ذخیره نام متغیرها (`strdup`) تخصیص داده شده بود را آزاد می‌کند تا از نشت حافظه (Memory Leak) جلوگیری شود.

```
153 int main(int argc, char **argv) {
154     printf("Starting interpreter...\n");
155     yyparse(); //parsing
156 
157     for (int i = 0; i < symbolCount; i++) {
158         if (symbolTable[i].name) {
159             symbolTable[i].name = NULL;
160         }
161     }
162     symbolCount = 0;
163 
164     return 0;
165 }
```

تابع main: نقطه شروع و پایان برنامه

نگاهی کلی به نحوه همکاری تمام قطعات با یکدیگر:

1. **`main`** برنامه را شروع می‌کند.
2. `main` تابع **`yyparse`** (از Bison) را فراخوانی می‌کند.
3. `yyparse` برای دریافت توکن‌ها، مکرراً تابع **`yylex`** (از Flex) را فراخوانی می‌کند.
4. `yyparse` با استفاده از **قوانین گرامر**، توکن‌ها را با الگوها تطبیق می‌دهد.
5. با تطبیق هر قانون، **کد C** مربوط به آن اجرا می‌شود (محاسبه، ذخیره متغیر و ...).
6. در صورت بروز خطای نحوی، **`yyerror`** فراخوانی می‌شود.
7. پس از پایان، `main` حافظه را پاک‌سازی می‌کند.

مثال:

```
int main() {
    let num_1 = 10;
    let num_2 = 20;
    let entry = vorood();
    let result = entry* (num_1 + num_2);
    show(result);
}
```

مرحله اول - تبدیل کد به توکن‌ها (Tokenization)

قبل از هر کاری، تحلیلگر لغوی (Flex) کل کد را می‌خواند و آن را به یک دنباله از "کلمات" یا همان **توکن‌ها** تبدیل می‌کند. هر توکن نوع و گاهی یک مقدار مشخص دارد.

جریان توکن‌های تولید شده:

- `INT_KEYWORD, MAIN_KEYWORD, LPAREN, RPAREN, LBRACE`
- `LET_KEYWORD, IDENTIFIER("num_1"), ASSIGN_OP, INTEGER(10), SEMICOLON`
- `LET_KEYWORD, IDENTIFIER("num_2"), ASSIGN_OP, INTEGER(20), SEMICOLON`
- `LET_KEYWORD, IDENTIFIER("entry"), ASSIGN_OP, INPUT_KEYWORD, LPAREN, RPAREN, SEMICOLON`
- `LET_KEYWORD, IDENTIFIER("result"), ASSIGN_OP, IDENTIFIER("entry"), MUL_OP, LPAREN, IDENTIFIER("num_1"), ADD_OP, IDENTIFIER("num_2"), RPAREN, SEMICOLON`
- `SHOW_KEYWORD, LPAREN, IDENTIFIER("result"), RPAREN, SEMICOLON`
- `RBRACE`

مرحله دوم - تجزیه و محاسبه عبارت (Parsing)

اینجا Bison (تجزیه‌گر) وارد عمل می‌شود. بیایید روی پیچیده‌ترین خط، یعنی `let result = entry * (num_1 + num_2);` تمرکز کنیم. Bison این عبارت را **از پایین به بالا** و **از درون پرانتز** تجزیه می‌کند.

- **محاسبه `(num_1 + num_2)`:**
  - تجزیه‌گر ابتدا `IDENTIFIER("num_1")` را می‌بیند، قانون `primary: IDENTIFIER` تطبیق می‌یابد و با `lookupSymbol("num_1")` مقدار **10** را به دست می‌آورد.
  - سپس `"num_2"(IDENTIFIER)` را با `lookupSymbol` به مقدار **20** تبدیل می‌کند.
  - حالا عبارت `primary(10) ADD_OP primary(2)` را می‌بیند. این با قانون `expression: expression ADD_OP term` مطابقت دارد.
  - عمل `{ $$ = $1 + $3; }` اجرا شده و نتیجه این `expression` داخلی عدد **30** می‌شود.

مرحله دوم - تجزیه و محاسبه عبارت (Parsing)

محاسبه `entry`:

- تجزیه‌گر `entry"(IDENTIFIER)"` را با `lookupSymbol` به مقداری که کاربر وارد کرده (فرض کنیم **5**) تبدیل می‌کند.
- اکنون عبارت `primary(5) MUL_OP primary(30)` را در اختیار دارد. (نتیجه پرانتز خودش یک `primary` است).
- این الگو با قانون `term: term MUL_OP factor` مطابقت دارد.
- عمل `{ $$ = $1 * 3; }` اجرا شده و حاصل `5 * 30` یعنی **150** به عنوان نتیجه نهایی کل `expression` به دست می‌آید.

مرحله سوم - ذخیره و نمایش نتیجه نهایی

پس از محاسبه مقدار نهایی عبارت، Bison به تجزیه بقیه دستورات ادامه می‌دهد.

- **ذخیره `result`:**
  - تجزیه‌گر اکنون الگوی کامل را می‌بیند: `LET_KEYWORD IDENTIFIER("result") ASSIGN_OP expression(150) SEMICOLON`
  - این با قانون `statement` تطبیق دارد و عمل `storeSymbol($2, $4);` اجرا می‌شود:
    - **`$2`** مقدار `IDENTIFIER` است: رشته `"result"`.
    - **`$4`** مقدار محاسبه‌شده `expression` است: عدد `150`.
  - در نتیجه `storeSymbol("result", 150)` فراخوانی شده و مقدار نهایی در جدول نمادها ذخیره می‌شود.

مرحله سوم - ذخیره و نمایش نتیجه نهایی

نمایش `result`:

- در نهایت، دستور `show(result);` تجزیه می‌شود.
- قانون `statement: SHOW_KEYWORD LPAREN expression RPAREN ...` تطبیق می‌یابد.
- در اینجا `expression` همان `IDENTIFIER("result")` است. با `lookupSymbol("result")` مقدار **150** بازیابی می‌شود.
- عمل `printf("%d\n", $3);` اجرا می‌شود:
  - **`$3`** مقدار `expression` است: عدد `150`.
- در نهایت، عدد **150** در کنسول چاپ می‌شود.

تشکر از توجهتون!