Compiladores
Roteiro de Laboratório 04 – Verificação de Tipos

1 Introdução

A tarefa deste laboratório é implementar um verificador de tipos (*type checker*) para a linguagem EZLang. Juntamente com a tarefa do Laboratório 03, a tarefa de hoje conclui o desenvolvimento do analisador semântico para a linguagem.

Conforme visto no conteúdo teórico do Módulo 04, a verificação de tipos é elemento fundamental do processo de análise semântica. Como praticamente todas as linguagens de alto nível são tipadas, sempre é necessário verificar se o uso dos tipos está de acordo com a semântica da linguagem. Essa verificação (*type checking*) ocorre tanto em linguagens estáticas quanto dinâmicas, diferindo apenas no momento em que ela ocorre: em tempo de compilação nas linguagens estáticas, e em tempo de execução nas linguagens dinâmicas.

A linguagem EZLang segue o modelo clássico de linguagens imperativas derivadas de Algol, com uma verificação de tipos totalmente estática. A semântica do sistema de tipos da linguagem é descrita a seguir.

2 O sistema de tipos de EZLang

O sistema de tipos de EZLang é muito simples, possuindo somente quatro tipos primitivos, sem tipos compostos ou estruturados (*structs*). Os tipos básicos da linguagem são **inteiro**, **real**, **Booleano** e ***string***. Só é possível declarar uma variável por linha e não é permitido a inicialização de variáveis junto com a declaração.

As operações aritméticas são as quatro fundamentais, entretanto o operador + é sobrecarregado (mais detalhes abaixo). A linguagem também possui dois operadores de comparação, um comando de atribuição e um comando para impressão de valores na tela.

2.1 Verificação e Conversão de Tipos

Valem as seguintes regras para o sistema de tipos de EZLang:

- Tipo inteiro sofre *widening* implícito para real. Não é permitido *narrowing*. (Veja *slides* da Aula 04 para as explicações desses termos.)
- O operador + é sobrecarregado. Os tipos resultantes são dados pela tabela abaixo (i – inteiro, r – real, b – Booleano, s – *string*):

+	i	r	b	s
i	i	r	i	s
r	r	r	r	s
b	i	r	b	s
s	s	s	s	s

  Assim, vemos pela tabela acima que em uma expressão como `2 + 4.2`, o operando da esquerda (`2`) sofre *widening* para real (`2.0`) e a expressão como um todo tem real como tipo resultante. É fácil perceber que o operador + é sobrecarregado: se quaisquer dos operandos for uma *string*, a expressão resultante também é uma *string*, e o operador

representa uma concatenação de *strings*. Além disso, também é possível "somar" valores Booleanos (de forma similar ao C), aonde `false` vale 0 e `true` vale 1. A soma de dois Booleanos é equivalente à operação lógica de OU (*or*).

- Os demais operadores aritméticos manipulam os tipos como abaixo (e – erro):

$-*/$	i	r	b	s
i	i	r	e	e
r	r	r	e	e
b	e	e	e	e
s	e	e	e	e

- Operadores de comparação *sempre* geram um Booleano como resultado. Entretanto, tais operadores só admitem tipos compatíveis, conforme a tabela abaixo (✓– OK, e – erro):

$<, =$	i	r	b	s
i	✓	✓	e	e
r	✓	✓	e	e
b	e	e	e	e
s	e	e	e	✓

- O comando de atribuição só admite tipos idênticos, exceto no caso de inteiro sofrendo *widening* para real, conforme a tabela abaixo:

:=	i	r	b	s
i	✓	e	e	e
r	✓	✓	e	e
b	e	e	✓	e
s	e	e	e	✓

  Como um exemplo de uma verificação que o seu compilador deve fazer, suponha um comando de atribuição como

  ```
  x := "Valor: " + 2;
  ```

  aonde a variável `x` foi previamente declarada com o tipo real. Esse exemplo possui um erro de tipos, pois o resultado da expressão da direita da atribuição (RHS – *right-hand side*) é do tipo *string*, enquanto que a variável do lado esquerdo (LHS – *left-hand side*) é do tipo real. Segundo a tabela acima, essa atribuição de tipos é inconsistente.
- Uma última verificação de tipos da linguagem requer que todas as expressões de teste dos comandos de `if` e `repeat` sejam do tipo Booleano.

2.2 Mensagens do verificador de tipos

O seu verificador de tipos deve exibir as seguintes mensagens.

Operadores binários com tipos incompatíveis. Se os operadores aritméticos, de comparação, ou de atribuição possuírem operandos incompatíveis, exiba a seguinte mensagem de erro no terminal:

```
SEMANTIC ERROR (XX): incompatible types for operator 'OP',
                     LHS is 'LT' and RHS is 'RT'.
```

Aonde `XX` é o número da linha do programa onde o erro foi detectado, `OP` é o operador binário da expressão com erro, e `LT` e `RT` são os tipos das sub-expressões da esquerda e direita, respectivamente.

Expressões de teste. Se as expressões de teste dos comandos de `if` e `repeat` não forem do tipo Booleano, exiba a seguinte mensagem de erro no terminal:

```
SEMANTIC ERROR (XX): conditional expression in 'OP'
                     is 'ET' instead of 'bool'.
```

Aonde `XX` é o número da linha do programa onde o erro foi detectado, `OP` é o comando que contém a expressão, e `ET` é o tipo inferido da expressão de teste.

Demais mensagens. As demais mensagens de erros léxicos, sintáticos e semânticos são as mesmas do Laboratório 03 e devem continuar sendo exibidas como antes.

3 Valores e ações semânticas

Você deve implementar a verificação de tipos com ações semânticas no `bison`, da mesma forma que no laboratório anterior. Entretanto, para entender como proceder, precisamos aprender melhor como utilizar as variáveis semânticas (variáveis $) no `bison`. As subseções seguintes são um misto de revisão do que foi visto nos laboratórios anteriores com informações novas.

3.1 Valores semânticos

Uma gramática livre de contexto manipula os *tokens* somente pelos seus tipos. Por exemplo, se o corpo de uma regra contém um símbolo terminal (*token*) do tipo 'constante inteira' (`INT_VAL`), isso quer dizer que *qualquer* constante inteira é gramaticalmente válida naquela posição. O valor exato da constante é irrelevante para o processo de análise sintática da entrada: se `x+4` é sintaticamente válida, então `x+1` ou `x+3989` também são igualmente válidas.

Mas o valor exato é muito importante para o significado (semântica) da entrada após a análise sintática. Um compilador é inútil se ele não consegue distinguir entre 4, 1, ou 3989 como constantes de um programa de entrada. Assim, cada símbolo da gramática (terminal ou não-terminal) no `bison` possui um *valor semântico*.

Um *token* possui um tipo definido na gramática, como `INT_VAL`, `ID` ou `SEMI` (`;`). O tipo é a única informação necessária para o *parser* decidir aonde um *token* pode aparecer, e como agrupá-lo com outros *tokens*. As regras da gramática formal desconhecem qualquer outra informação sobre *tokens* exceto os seus tipos.

O valor semântico contém o restante das informações sobre o significado do *token*, tal como o valor de um inteiro, ou o nome de um identificador. Por outro lado, um *token* como `;` (`SEMI`) é somente pontuação e não precisa ter nenhum valor semântico.

Por exemplo, um *token* de entrada pode ser classificado como o tipo `INT_VAL` e ter o valor semântico `4` associado. Outro *token* pode ter o mesmo tipo mas o valor `3989`. Quando uma regra da gramática diz que um `INT_VAL` é esperado, qualquer um desses *tokens* serve porque ambos são do tipo `INT_VAL`. **Quando o *parser* aceita o *token*, ele mantém o registro do valor semântico do *token*.** (Note que para que um *token* como `INT_VAL` receba o seu valor semântico correto, o *scanner* deve preencher a variável `yylval` adequadamente. Veja o Exemplo 03 do Laboratório 02 para relembrar.)

Símbolos não-terminais da gramática também podem possuir um valor semântico. Por exemplo, em uma calculadora, uma expressão tipicamente possui um número como valor semântico (o valor da avaliação da expressão). Em um compilador, os valores semânticos das expressões podem formar uma árvore que descreve o significado da expressão.

3.2 Ações semânticas

Para um compilador ser útil ele precisa fazer mais do que realizar o *parsing* da entrada, ele também precisa produzir uma saída baseado na entrada fornecida. Em uma gramática do `bison`, uma regra pode ser associada a uma *ação semântica* formada por código C. A cada vez que o *parser* reconhece o corpo de uma regra (operação de *reduce*, veja os *slides* da Aula 02), a ação semântica associada é executada.

Na maioria das vezes, o propósito de uma ação é computar o valor semântico da construção toda (cabeça da regra) a partir dos valores semânticos dos elementos do corpo da regra. Por exemplo, suponha uma regra que define uma expressão de soma, como já visto várias vezes. Quando o *parser* reconhece a soma, as duas sub-expressões já possuem valores semânticos associados a ela. (Lembre que o `bison` constrói *parsers bottom-up*.) A ação semântica da regra de soma então deve criar um novo valor semântico para a expressão completa que acabou de ser reconhecida.

Por exemplo, a regra abaixo implementa uma calculadora de somas.

```
expr: expr PLUS expr   { $$ = $1 + $3; } ;
```

Em ações semânticas, nós podemos acessar os valores semânticos associados a cada símbolo da regra gramatical através das *variáveis semânticas* (variáveis `$`). A cabeça da regra é associada à variável `$$` e os símbolos do corpo da regra são associados às variáveis `$1`, `$2`, etc. Assim, as variáveis da regra acima ficam como abaixo.

```
expr: expr PLUS expr ;
----  ---- ---- ----
 $$    $1   $2   $3
```

É essencial notar que *todos* os símbolos do corpo da regra possuem uma variável associada, no entanto, nem todos os valores dessas variáveis devem ser utilizados. No exemplo acima, o *token* `PLUS` não possui um valor semântico, ele é somente o símbolo que representa a soma, e por isso não devemos utilizar o valor `$2` na ação. **Isso é um ponto muito importante, pois o uso incorreto das variáveis semânticas é fonte de muito erros difíceis de se *debugar*. Fique atento!**

É possível incluir ações semânticas no meio do corpo de uma regra, mas é essencial notar que essa ação **também recebe uma variável semântica**! Tomemos como exemplo a solução do Laboratório 03. Foi utilizada uma regra como abaixo, com as variáveis `$` indicadas.

```
assign-stmt: ID { check_var(); } ASSIGN expr SEMI { ... };
-----------  -- ---------------- ------ ---- ----
    $$       $1        $2          $3    $4   $5
```

Assim, se quisermos acessar o valor semântico da expressão na ação ao final da regra, devemos lembrar de usar `$4`, pois a ação no meio da regra também conta. **Esse é um erro muito comum entre os alunos. Cuidado!**

3.3 Exemplo 01 – Implementando uma calculadora

Podemos modificar a gramática do Exemplo 05 do Laboratório 02 (*parser* das quatro expressões aritméticas) para implementar uma calculadora básica. O trecho principal do arquivo `parser.y` é listado abaixo. (Veja o arquivo `CC_Lab04_Exemplos.zip`, no AVA.)

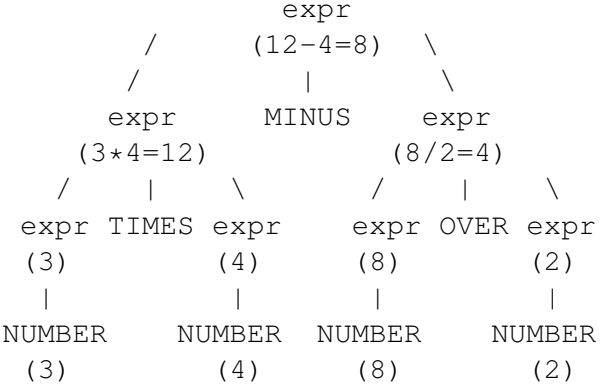
```
line:
  expr ENTER        { printf("Result = %d\n", $1); }
;

expr:
  expr PLUS expr    { $$ = $1 + $3; }
| expr MINUS expr   { $$ = $1 - $3; }
| expr TIMES expr   { $$ = $1 * $3; }
| expr OVER expr    { $$ = $1 / $3; }
| NUMBER            { $$ = $1; }
;
```

Compilando e executando o programa, obtemos algo como abaixo.

```
$ ./a.out <<< "3*4-8/2"
Result = 8
```

ATIVIDADE 0: remova os comandos `%left` da gramática para reintroduzir os conflitos de *shift-reduce* e teste novamente o comando acima. O resultado retornado será 0. Isso indica que o `bison` resolveu o conflito de forma errada. É por isso que dizemos que conflitos de *shift-reduce* são **erros** (embora o `bison` liste como *warnings*) pois o seu *parser* pode não ter o comportamento correto quando a gramática possui conflitos.

Para entendermos como as ações semânticas listadas acima funcionam, devemos pensar na construção *bottom-up* da *parse tree*, ilustrada abaixo.



Cada nó da árvore possui um símbolo da derivação da entrada. Abaixo de cada símbolo, entre parênteses, está listado o valor semântico associado àquele nó. A construção da árvore vai das folhas para raiz, assim, a medida que a árvore cresce (para cima) as ações semânticas vão calculando os valores a partir das sub-árvores já calculadas. O valor da expressão raiz é o número exibido na tela.

Você já deve ter percebido que para essa construção funcionar, o *token* `NUMBER` já deve possui um valor semântico associado, pois a ação semântica `{$$ = $1;}` assume essa situação. O responsável por introduzir esses valores nas folhas é o *scanner*, como visto abaixo.

```
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
```

Certifique-se que você entendeu completamente o exemplo acima antes de continuar.

4 Implementado o verificador de tipos

Para poder implementar o verificador de tipos de EZLang, você deve ser capaz de atribuir tipos adequados para todas as expressões da gramática, seguindo as regras de tipagem estabelecidas na Seção 2. A computação (inferência) dos tipos das expressões deve ser feita de forma recursiva, em um processo *bottom-up* igual ao do Exemplo 01. A diferença é que em vez de calcular os valores numéricos das expressões, você vai inferindo e propagando os tipos adequadamente, até o momento de fazer as verificações de compatibilidade. (Veja exemplos nos *slides* da Aula 04.)

Como descrito acima, se é necessário determinar o tipo de cada (sub) expressão, então as variáveis semânticas devem carregar a informação de tipo. Por padrão, essas variáveis no `bison` são do tipo inteiro, mas nós queremos modificar esse tipo para usarmos a enumeração `Type` definida no arquivo `types.h` (veja Laboratório 03). Para tal, utilize o comando abaixo imediatamente antes das definições dos *tokens* no `bison`.

```
%define api.value.type {Type}
```

ATIVIDADE 1: Modifique a sua solução do laboratório anterior para realizar a verificação de tipos conforme descrita nesse documento.

Algumas observações importantes:

- O seu compilador pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- Os programas de entrada para teste são os mesmos de sempre (`in.zip`). As saídas esperadas desta tarefa estão no arquivo `out04.zip`, disponível no AVA.