Compiladores
Roteiro de Laboratório 05 – Construção de ASTs

1 Introdução

A tarefa deste laboratório é implementar a construção de uma representação intermediária (IR – *intermediate representation*) do código de entrada para posterior execução. Conforme discutido nas aulas, a IR é a "ponte" de comunicação entre o *front-end* e o *back-end* do compilador. Juntamente com as tarefas dos laboratórios anteriores (analisadores léxico, sintático e semântico), a tarefa de hoje conclui o desenvolvimento do *front-end* de EZLang.

Vamos utilizar como IR uma construção bastante comum, a AST (*Abstract Syntax Tree* – Árvore de Sintaxe Abstrata). Como visto anteriormente, a AST é uma versão simplificada (abstrata) da *parse tree*, também chamada de CST (*Concrete Syntax Tree* – Árvore de Sintaxe Concreta). As duas árvores diferem fundamentalmente na informação (representação) que elas carregam. Enquanto a CST indica a derivação gramatical (sintática) do código de entrada, a AST é a versão "enxuta" da CST, contendo somente a estrutura do código que é necessária para sua execução. (Veja o material sobre *parsing*, para uma revisão desses conceitos.)

2 Implementação da AST

Essa seção descreve a implementação da AST fornecida no arquivo `CC_Lab05_src.zip`. Cada nó da AST possui a seguinte estrutura (veja o arquivo `ast.c`):

- `kind`: o comando ou estrutura representado pelo nó. Essa representação varia desde construtos simples, como uma constante inteira (`INT_VAL_NODE`), até estruturas mais elaboradas, como um bloco de código (`BLOCK_NODE`). Todos os `kind`s da AST serão descritos adiante.
- `data`: um dado adicional sobre o nó que pode ou não ser utilizado. O significado desse campo varia conforme o `kind` do nó, como descrito na próxima subseção. Como esse campo pode ser utilizado para armazenar dados de tipos distintos (inteiros ou reais), empregamos uma `union` para economizar memória. (Se achar necessário, veja uma breve explicação sobre `union`s em C aqui: `https://www.tutorialspoint.com/cprogramming/c_unions.htm`.)
- `type`: é o tipo da expressão representada pelo nó. Por exemplo, uma operação de comparação tem o tipo `BOOL_TYPE` (veja `types.h`). Já comandos de declarações (*statements*) não possuem tipo (`NO_TYPE`). Essa informação de tipo já foi computada no laboratório anterior e deve agora ser movida para dentro da AST. (Mais detalhes adiante.)
- `count`: número de filhos do nó.
- `child`: lista de ponteiros para os nós filhos. Como já é de costume nas aulas de laboratório, a implementação desse lista é bastante miserável, com um limite máximo de filhos por nó *hard coded* em um `#define`. Obviamente essa construção **nunca** deve ser usada em uma aplicação real e você é livre para implementar uma AST adequada, aonde os nós admitem um número arbitrário e variável de filhos. No entanto, deixe para fazer essa modificação **depois** de terminar a tarefa desse laboratório. No momento, estamos mais interessados no **uso** da AST e não na sua implementação.

Uma descrição detalhada de todos os `kind`s dos nós da AST é dada a seguir.

2.1 Descrição dos nós da AST

A enumeração `NodeKind` definida no arquivo `ast.h` lista todos os possíveis `kind`s dos nós da AST. Temos 20 construções derivadas diretamente da estrutura da linguagem EZLang, e mais 6 `kind`s que indicam conversões de tipos. Vamos agora discutir detalhadamente cada possível construção dos nós. (*Obs.:* Quando o campo `data` não for mencionado na descrição de um `kind` específico, quer dizer que ele não é utilizado nesse nó.)

- `ASSIGN_NODE`: nó que representa um comando de atribuição. Possui dois filhos, o primeiro é um `VAR_USE_NODE` que indica a variável atribuída, e o segundo é uma expressão com um tipo compatível (conforme as regras definidas e implementadas no laboratório anterior). Caso a expressão precise sofrer *widening* de inteiro para real, é necessário criar um nó de conversão (`I2R_NODE`) para o segundo filho. Como uma atribuição em EZLang não é uma expressão (ao contrário de C), esse nó não tem tipo (`NO_TYPE`).
- `EQ_NODE`: nó que representa uma expressão de comparação (igualdade). Possui dois filhos, correspondentes às sub-expressões da esquerda e direita, respectivamente. Pode ser necessário criar um nó de conversão (`I2R_NODE`) para um dos filhos, no caso da sub-expressão precisar sofrer *widening* de inteiro para real. (Novamente, todas as informações de tipagem e conversões foram estabelecidas no laboratório anterior.) Esse nó tem o tipo Booleano (`BOOL_TYPE`).
- `BLOCK_NODE`: nó que representa um bloco de código, correspondente ao não-terminal `stmt-list` da gramática. Pode indicar tanto a sequência de comandos do programa, quanto os blocos dentro de `if`s e `repeat`s. O número e a sequência de filhos do nó devem corresponder à quantidade e à sequência dos comandos do bloco, de forma que uma visitação linear na lista de filhos equivale exatamente à execução sequencial dos comandos do bloco. (*Obs.:* Esse nó destaca imediatamente a limitação da implementação da AST, que só permite um bloco com no máximo 20 comandos – número máximo de filhos de um nó – quando o correto seria permitir um número arbitrário de filhos.) Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.
- `BOOL_VAL_NODE`: nó que representa uma constante Booleana, tendo o tipo `BOOL_TYPE`. O campo `data` é utilizado nesse nó, contendo os valores `0` ou `1` para indicar os Booleanos `false` ou `true`, respectivamente. Esse é um nó folha na AST, podendo ser criado diretamente pelo *scanner*.
- `IF_NODE`: nó que representa um comando `if` no código. Possui pelo menos dois filhos, com um terceiro opcional. O primeiro filho é a expressão de teste, devendo necessariamente ser do tipo `BOOL_TYPE`. O segundo filho é o bloco de comandos do `then`, e o terceiro representa o bloco do `else` quando existir. Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.
- `INT_VAL_NODE`: nó que representa uma constante inteira, tendo o tipo `INT_TYPE`. O campo `data` é utilizado nesse nó para armazenar o valor inteiro do lexema reconhecido. Esse é um nó folha na AST, podendo ser criado diretamente pelo *scanner*.
- `LT_NODE`: nó que representa uma expressão de comparação (menor que). Valem os mesmos comentários feitos para o nó `EQ_NODE` acima.
- `MINUS_NODE`: nó que representa uma expressão de subtração. Possui dois filhos, correspondentes às sub-expressões da esquerda e direita, respectivamente. Pode ser necessário criar um nó de conversão (`I2R_NODE`) para um dos filhos, no caso da sub-expressão precisar sofrer *widening* de inteiro para real. O tipo do nó pode ser `INT_TYPE` ou `REAL_TYPE`, conforme a semântica do sistema de tipos do laboratório anterior.
- `OVER_NODE`: nó que representa uma expressão de divisão. Valem os mesmos comentários feitos para o nó `MINUS_NODE` acima.

- `PLUS_NODE`: nó que representa o símbolo sobrecarregado +. Possui sempre dois filhos, como os demais operadores binários. Para operandos numéricos, representa a operação aritmética de soma. Para operandos Booleanos, equivale à operação lógica OU. Já para *strings*, representa a operação de concatenação. O tipo do nó é determinado pela semântica do sistema de tipos. Note que para várias combinações de tipos dos operandos pode ser necessário cria um nó de conversão para um dos filhos: `I2R_NODE`, `I2S_NODE`, etc.
- `PROGRAM_NODE`: nó raiz da AST, representando o programa como um todo. Possui dois filhos, o primeiro é um `VAR_LIST_NODE` contendo as variáveis declaradas, e o segundo é um `BLOCK_NODE` contendo os comandos do programa. Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.
- `READ_NODE`: nó que representa um comando de entrada de dados. Possui um único filho `VAR_USE_NODE`, aonde o tipo da variável determina o tipo do valor a ser lido. (Pode-se dizer que o comando é sobrecarregado.) Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.
- `REAL_VAL_NODE`: nó que representa uma constante real, tendo o tipo `REAL_TYPE`. O campo `data` é utilizado (como um `float`) nesse nó para armazenar o valor do lexema reconhecido. Esse é um nó folha na AST, podendo ser criado diretamente pelo *scanner*.
- `REPEAT_NODE`: nó que representa um comando `repeat` no código. Possui dois filhos, o primeiro é o bloco de comandos do *loop* e o segundo é a expressão de teste, devendo necessariamente ser do tipo `BOOL_TYPE`. Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.
- `STR_VAL_NODE`: nó que representa uma *string* constante, tendo o tipo `STR_TYPE`. O campo `data` é utilizado nesse nó para armazenar o índice da constante na tabela de *strings*. (Em uma implementação mais realista da tabela, como em um *hash* por exemplo, devemos utilizar um ponteiro para o *bucket* ao invés de um índice direto. Nesse caso, o campo `data` precisaria ser expandido para suportar ponteiros.) Esse é um nó folha na AST, podendo ser criado diretamente pelo *scanner*.
- `TIMES_NODE`: nó que representa uma expressão de multiplicação. Valem os mesmos comentários feitos para o nó `MINUS_NODE` acima.
- `VAR_DECL_NODE`: nó que representa uma declaração de variável no início do programa. O campo `data` é utilizado nesse nó para armazenar o índice da variável na tabela de variáveis. (Valem os mesmos comentários de implementação feitos para `STR_VAL_NODE`.) Esse é um nó folha na AST, mas ele deve ser criado pelo *parser*, pois são necessárias algumas verificações semânticas (Laboratório 03). O tipo do nó corresponde ao tipo da variável declarada.
- `VAR_LIST_NODE`: nó que representa a lista de todas as variáveis declaradas no programa, cujo tamanho varia conforme a quantidade de linhas da seção `var`. Recomenda-se sempre criar esse nó mesmo quando o programa não possui variáveis (zero filhos), pois isso facilita o caminhamento na AST depois. (O nó sempre existe, então não é preciso testar para ponteiros nulos.) Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.
- `VAR_USE_NODE`: nó que representa a utilização de uma variável em qualquer ponto do programa. Valem os mesmos comentários feitos para o nó `VAR_DECL_NODE` acima.
- `WRITE_NODE`: nó que representa um comando de saída. Possui um único filho que é a expressão que deve ser exibida em `stdout`. Assim como o nó `READ_NODE`, esse comando pode ser considerado sobrecarregado, aonde o tipo da expressão determina como deve ser realizada a impressão. Esse nó não tem tipo (`NO_TYPE`) por não ser uma expressão.

Além dos 20 `kind`s de nós descritos acima, a AST admite mais 6 nós para conversão de tipos: `B2I_NODE`, `B2R_NODE`, `B2S_NODE`, `I2R_NODE`, `I2S_NODE` e `R2S_NODE`, cuja semântica deve ser evidente pelos nomes dos nós. Nós de conversão possuem exatamente um filho, corres-

pondente à expressão que deve ser convertida. O tipo do nó é o tipo alvo da conversão. Por exemplo, um `B2S_NODE` é do tipo *string* e possui um filho com tipo Booleano.

3 Implementado a construção da AST

A construção da AST no seu *front-end* deve ser feita por meio de ações semânticas dentro do *parser*, de forma recursiva, em um processo *bottom-up* igual ao que foi feito para verificação e inferência de tipos no Laboratório 04. Para cada regra da gramática de EZLang, você deve incluir uma ou mais ações semânticas que constroem sub-árvores da AST, até se chegar na raiz.

No laboratório anterior, utilizamos as variáveis semânticas para carregar as informações de tipo que eram inferidas. Agora, vamos utilizar essas mesmas variáveis para armazenar ponteiros para sub-árvores da AST, utilizando-os para conectar recursivamente as sub-árvores até que a AST fique completa. Assim, devemos trocar o tipo das variáveis semânticas do `bison` com o comando abaixo.

```
%define api.value.type {AST*}
```

Essa mudança explica porque os nós da AST possuem um campo de tipo: as informações de tipagem que antes eram armazenadas diretamente nas variáveis semânticas agora ficam guardadas nos nós da AST. A verificação de tipos e a construção da AST devem ser feitas simultaneamente, de forma que se a verificação semântica passar, o programa de entrada está correto e a AST correspondente é construída.

3.1 Visualizando as ASTs

O processo de construção da AST é complexo, e portanto, propenso a erros. Para facilitar um pouco a vida do programador, a implementação da AST fornecida neste laboratório provê um formato de saída na linguagem `dot` da ferramenta GraphViz. Seja o programa de entrada abaixo, no arquivo `c02.ezl`.

```
program addone;
var
int x;
begin
    read x;
    x := x + 1;
    write x;
end
```

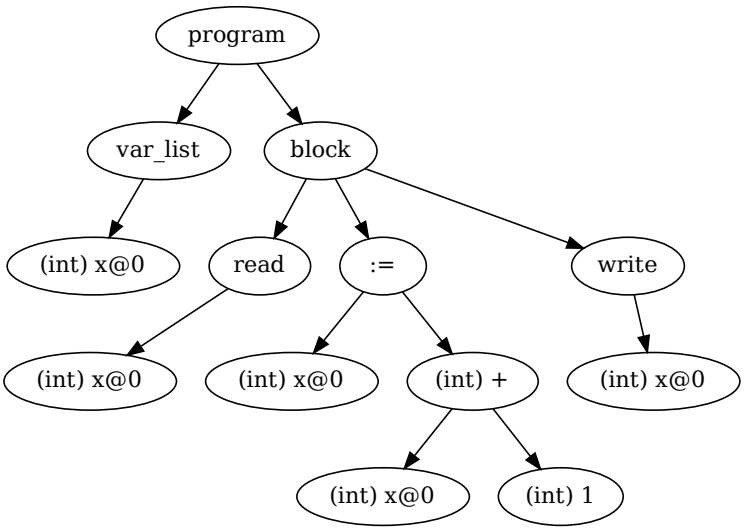
Para visualizar a árvore desse programa, é preciso chamar a função `print_dot`, passando o ponteiro para a raiz da AST a ser exibida. Essa função normalmente é chamada após o término do processo de *parsing* para mostrar a AST construída. Após compilar o executável deste laboratório, podemos utilizar redirecionamentos no terminal, como a seguir. (Para evitar a manipulação de arquivos, a impressão do formato `.dot` é feita em `stderr`.)

```
$ ./lab05 < c02.ezl 2> c02.dot
```

No comando acima, a saída normal do compilador continua sendo exibida em `stdout` e a saída de `stderr` é redirecionada para o arquivo `.dot`. Por fim, podemos gerar um arquivo PDF com o comando abaixo.

```
$ dot -Tpdf c02.dot -o c02.pdf
```

O resultado será um arquivo PDF contendo a figura a seguir.



Essa é a representação visual da AST para o programa de entrada. As seguintes notações foram utilizadas na impressão dos nós:

- Para todos os nós com um tipo, este é prefixado entre parênteses. Ex.: `(int)+` indica um nó de soma de inteiros.
- Nós de declaração ou uso de variáveis exibem o identificador seguido do índice na tabela de símbolos. Ex.: `(int) x@0` indica uma variável `x` armazenada no índice 0 da tabela.
- Constantes numéricas (inteiras e reais) são acompanhadas pelo seu tipo. Constantes Booleanas são impressas com a sua representação interna (0 ou 1).
- *Strings* constantes são indicadas pelo seu índice na tabela de *strings*.

3.2 Tarefa

Modifique a sua solução do laboratório anterior para realizar a construção da AST conforme descrito nesse documento.

Algumas observações importantes:

- O seu compilador pode terminar a execução ao encontrar o primeiro erro no programa de entrada.
- As mensagens de erros léxicos, sintáticos e semânticos são as mesmas do Laboratório 04 e devem continuar sendo exibidas como antes.
- Os programas de entrada para teste são os mesmos de sempre (`in.zip`). As saídas esperadas desta tarefa estão no arquivo `out05.zip`, disponível no AVA.