Projeto de Compiladores:
`Pico`

Nicolas Maillard Stéfano D. K. Mór Sílvio Cordeiro

Este documento detalha a especificação do pequeno compilador `Pico`.

Organização do projeto, *deadlines* e considerações gerais

Organização

A etapa 1 consiste na programação de um analisador lexical. A segunda etapa levará a implementar uma estrutura de dados de árvore, e o analisador sintático que a usará. As etapas 3, 4 e 5 tratarão da fase de geração de código, ao decorar a árvore com atributos semânticos apropriados.

1. Analisador lexical (com auxílio do flex). Peso 1. Entrega até 29 de Agosto, 22h00.
2. Estruturas de dados de árvore e parser. Peso 2. Entrega até 21 de Setembro, 22h00.
3. Análise semântica: declaração de variáveis e tratamento de expressões. Peso 2. Entrega até 09 de Novembro, 22h00.
4. Análise semântica: arrays. Peso 2. Entrega até 21 de Novembro, 22h00.
5. Análise semântica: laços. Peso 3. Entrega até 12 de Dezembro, 22h00.

A programação deve ser feita no Linux, em C ANSI. Serão usadas as ferramentas Lex (flex) e Yacc (Bison). O último capítulo deste documento dá as primeiras instruções para usar o comando Make, no Linux, para compilar um programa. Também explica como se usa o Doxygen para documentar código.

É absolutamente fundamental que a especificação seja respeitada totalmente: os nomes dos procedimentos devem ser iguais ao especificado; os parâmetros devem ser conformes ao que se espera; etc. Serão usados testes automatizados (graças a scripts), os quais apenas poderão ser executados caso a especificação seja respeitada. Os códigos devem ser comentados, de acordo com o padrão Doxygen. Encontra-se, nos programas entregues junto a essa especificação, exemplos de uso de Doxygen.

O projeto todo deve ser desenvolvido por grupos de 3 alunos. Faz parte do trabalho se distribuir a carga de trabalho de forma a fornecer uma implementação de boa qualidade: por exemplo, pode se "programar a quatro mãos" (dois alunos programam juntos), com o terceiro colega se encarregando de testes e da documentação. Pode-se também optar por uma distribuição dos procedimentos a ser programados. Aconselha-se que programação e teste não sejam feitos pela mesma pessoa. É interditado que um aluno nunca programe nada. É totalmente possível que os integrantes de um grupo tenham notas diferenciadas em função de seu envolvimento no trabalho.

Protocolo de entrega

Para cada etapa, haverá um encontro em laboratório durante uma aula (terça- ou quinta-feira, às 10h30), para apresentação e discussão da implementação. Esses encontros estão agendados no Moodle da disciplina. No dia anterior ao encontro, o código fonte deve ser disponibilizado aos tutores até às 22h00. São

esses dias de *entrega* do código que são informados acima junto com o intitulado de cada etapa. Verifica-se no Moodle que esses dias de entrega são vésperas de encontros presenciais.

A entrega do código se fará da forma seguinte:

- Todo o trabalho de programação deve se encontrar na pasta `Pico/src/` do programa fonte original liberado. Pode-se alterar os Makefiles providos em função de suas necessidades, porém as compilações previstas neles devem ser mantidas funcionais. É recomendado usar o diretório `Tests/` para efetuar (e armazenar) os testes de suas estruturas de dados. Os testes executados automaticamente pelos professores serão executados no diretório `Tests/`.
- Renomear (ou copiar recursivamente...) `Pico/` em `Pico-Beltrano-Ciclano-Fulano-etapaX`, onde o `X` final será o número da etapa entregue ($X = 1, 2, \ldots, 5$). A seguir, e para exemplificar, usou-se $X = 2$. **Importante:** mantenham os nomes e a ordem dos nomes `Beltrano-Ciclano-Fulano` iguais em todas as etapas.
- Esse diretório deve ser arquivado e compactado através do uso da ferramenta `tar` do Linux, pelo comando:

  ```
  tar cvzf Pico-Beltrano-Ciclano-Fulano-etapa2.tgz \
     Pico-Beltrano-Ciclano-Fulano-etapa2/.
  ```

  Como resultado, deve-se obter um arquivo chamado `Pico-Beltrano-Ciclano-Fulano-etapa2.tgz`.
- Deve-se copiar `Pico-Beltrano-Ciclano-Fulano-etapa2.tgz` no diretório `public_html` do *home*, na máquina html.inf.ufrgs.br, de um dos três alunos do grupo (por exemplo: Beltrano), e possibilitar seu acesso em leitura: (`chmod a+r Pico-Beltrano-Ciclano-Fulano-etapa2.tgz`).

  A partir daí, qualquer usuário deve conseguir baixar seu trabalho, ao apontar um navegador Web para a URL `http://www.inf.ufrgs.br/~beltrano/Pico-Beltrano-Ciclano-Fulano-etapa2.tgz`. É aconselhado efetuar um teste para verificar que o *download* é autorizado.
- Mandar um email para o tutor (nicolas@inf.ufrgs.br, srcordeiro@inf.ufrgs.br), para indicar a URL `http://www.inf.ufrgs.br/~beltrano/Pico-Beltrano-Ciclano-Fulanoo-etapa2.tgz` até as 22h00. Este email deve ser mandado a cada etapa, informando a nova URL. Pode ser informada mais de uma URL; neste caso, a versão mais recente será baixada para testes.

MUITO IMPORTANTE: os tutores vão usar *scripts* automatizados para recuperar e testar os trabalhos. Qualquer falha ao seguir as regras acima descritas (uso de outras ferramentas de compressão, esquecer de atualizar o direito de leitura, alteração nos nomes de arquivos...) irá levar à impossibilidade de recuperar o trabalho e a uma nota (parcialmente) zerada.

Avaliação

A avaliação se fará em duas etapas.

Primeiramente, seu programa irá ser testado de forma automatizada: programas C preescritos irão ser compilados junto com suas implementações, e executar operações usuais nelas para verificar sua conformidade à especificação (exemplos de tais testes serão providos nessa semana). A conformidade irá levar a 50% da nota final obtida nessa etapa. Salienta-se que, nessa fase, programas de verificação de cópia e programas de busca por código on-line serão usados para detectar eventuais fraudes.

Após o encerramento do prazo de entrega, os grupos terão mais 24 horas para entregar, opcionalmente, uma nova versão que corrija *bugs*, sejam os mesmos detectados pelos alunos, sejam eles encontrados pelos testes automatizados ou na sessão de encontro. Se houver entrega de uma nova versão neste prazo de 24h, a mesma será testada pelos tutores, e a nova nota de testes será considerada, porém não podendo exceder 8/10.

A outra metade da nota será atribuída em função da apresentação informal que vocês farão em laboratório, no dia seguinte, ao seu tutor: resultados dos testes, discussão da implementação, validação através de outros testes, documentação. Todos os três alunos podem ser indagados sobre qualquer parte da implementação, até mesmo quem não programou uma funcionalidade: espera-se que todos os integrantes tenham entendido como tudo foi programado. Opcionalmente, o tutor poderá aplicar um questionário rápido, e escrito, para testar o conhecimento do programa entregue.

Qualquer tentativa, mesmo parcial, de reaproveitamento de código de colegas ou da Web, irá zerar a nota.

Chapter 1

Especificação da Etapa 1

Essa primeira etapa do projeto consiste na implementação do analisador lexical (scanner), que deverá retornar os tokens reconhecidos num arquivo de entrada.

1.1 Explicações

A etapa consiste na definição, com o Flex, dos tokens que serão reconhecidos pelo compilador `Pico`. Cada token será implementado, em C, por um número constante inteiro através de diretivas `#define`. Por exemplo, `#define IDF 101` pode servir para definir o token `IDF`, que será encodificado no analisador lexical (e no futuro analisador sintático) através da constante 101.

A lista inteira dos tokens que deverão ser definidos, junto com a especificação (em português) dos lexemas que deverão ser reconhecidos como representando esses tokens, se encontra a seguir na próxima seção.

Para cada token, você deverá:

1. Defini-lo através de um `#define`. Essa lista de `define` deverá ser programada em um arquivo separado, limitado a este conteúdo, chamado de `tokens.h`. O seu analisador lexical deverá incluir `tokens.h` através de um:
   `#include ''tokens.h''`

   Os arquivos entregues em `Pico/src/` incluem um esqueleto para `tokens.h` e a inclusão do mesmo no arquivo de entrada usado com o Flex (chamado `scanner.l`).

2. Definir uma expressão regular, de acordo com a sintaxe do Flex, que especifique os lexemas representando esses tokens. Essas expressões regulares deverão se encontrar no arquivo `scanner.l`, usado como entrada do Flex. O arquivo `scanner.l` inicial, encontrado em `Pico/src/`, contém um esqueleto "pronto a compilar". Sugere-se ler a apostila sobre o (F)lex para determinar a sintaxe exata das expressões regulares usadas no Flex.

3. Associar a essa expressão regular uma ação em C, que será acionada quando o analisador lexical irá reconhecer este token. A ação, em geral, será trivial e consistirá apenas no retorno (`return`) do token identificado. Eventualmente, será necessário executar uma ação mais complexa, de acordo com a especificação. Caberá, então, programar (em C) o código necessário ou usar estruturas de dados já prontas.

Exemplo: seja a especificação (irrealista) seguinte: o token `INT` deve ser retornado ao reconhecer os lexemas `integer` ou `int`. Deve-se programar, por exemplo:

```
#define INT 3333
%%
(integer|int)    { return( INT ); }
```

A linha do `#define` irá no arquivo `tokens.h`. A linha com a expressão regular, simples neste caso, e a ação em C (`return()`), deverão ser inseridos no arquivo `scanner.l`.

Seu analisador lexical, chamado `pico`, deverá ler sua entrada a partir de um arquivo especificado em argumento na linha de comando.

O arquivo `scanner.l` e o Makefile fornecidos em `Pico/src` são pontos de partida para escrever o analisador lexical e compilá-lo. Basicamente, basta complementar `scanner.l` e `tokens.h`. `make` ou `make pico` deve invocar o flex, gerar o arquivo C que implementa o analisador, e compilá-lo para obter o executável `pico`. Observa-se que o `scanner.l` já vem com um `main` pronto, o qual se encarrega de ler o arquivo em entrada e de chamar o scanner nele.

1.2 Especificação dos tokens

Os tokens são definidos a seguir em duas partes: os chamados "tokens simples" representam apenas um lexema (string) único, o que significa que a expressão regular os definindo é trivial (mas precisa ser implementada). Os outros tokens necessitam de expressões regulares menos imediatas, porém todas discutidas nas aulas.

1.2.1 Tokens simples

Cada um dos lexemas a seguir se confunde com o token a ser retornado, ou seja o token representa apenas um única lexema. A única ação a ser efetuada consiste no retorno do token.

Têm casos especiais ainda mais simples, onde o lexema se limita a um caractere único (exemplos: `*`, ou `;`). Neste caso, o Flex possibilita retornar como constante inteira o próprio caractere (sem necessitar um `#define`, pois será usado um *cast* do *char* de C para um *int*). Por exemplo, se o lexema `*` deve ser reconhecido e associado a um token, ao invês de chamá-lo, por exemplo, de `ASTERISCO` e de programar seu retorno assim:

```
#define ASTERISCO 3333
%%
''*''    { return( ASTERISCO ); }
```

basta escrever:

```
%%
''*''    { return( '*' ); }
```

A tabela a seguir explicita todos os tokens a serem retornados. Para ganhar espaço, alguns casos juntam mais de um token numa linha só, separados por vírgulas. Espera-se bom senso para entender que, por exemplo,

Lexema	Token a ser retornado
`(,)`	'(', ')' (respectivamente)

significa que o lexema `(` deve ser associado ao token '(' (exemplo do caso onde o lexema se limita a um único caractere) e que o lexema `)` deve ser associado ao token ')', e NÃO que o lexema `(, )` (string composto por "abre-parêntese vírgula fecha-parêntese") deve ser associado aos dois tokens '(' e ')' (o que não faria sentido).

Lexemas	Tokens a ser retornados
`int`	INT
`double`	DOUBLE
`real`	REAL
`char`	CHAR
`string`	STRING
`*,+,-,/`	'*', '+', '-', '/' (respectivamente)
`,`	','
`;`	';'
`:`	':'
`'`	QUOTE
`(,)`	'(', ')' (respectivamente)
`[,],{,}`	'[', ']', '{', '}' (respectivamente)
`<,>,=`	'<', '>' e '=' respectivamente
`<=`	LE
`>=`	GE
`==`	EQ
`<>`	NE
`&,|,!`	AND, OR, NOT (respectivamente)
`if`	IF
`then`	THEN
`else`	ELSE
`for`	FOR
`next`	NEXT
`while`	WHILE
`end`	END
`true`	TRUE
`false`	FALSE

1.2.1.1 Outros tokens

Em toda essa seção:

- um dígito é um dos caracteres '`0`', '`1`', '`2`', ..., '`9`';
- um espaço branco é o caractere obtido quando se aperta a tecla "espaço".

Descrição	token a ser retornado	Ação
Qualquer combinação de um ou mais "brancos": espaço branco, tabulação, fim de linha.	não será retornado token	imprimir na saída (tela): `SPACE`.
Pelo menos uma letra minúscula ou um _ (*underscore*), seguido por qualquer número de letras (maiúsculas ou minúsculas), _ (*underscore*) ou dígitos.	IDF	O lexema deve ser impresso na tela.
Pelo menos uma letra maiúscula, seguida de qualquer sequência de _ (*underscore*) ou letras maiúsculas ou dígitos.	CONST	O lexema deve ser impresso na tela.
Qualquer sequência de caracteres (letras minúsculas ou maiúsculas, dígitos, espaços brancos. Qualquer outro caractere é excluído), começando e terminando por aspas `"`.	STR_LIT	O lexema (incluindo as aspas) deve ser impresso na tela .
Qualquer conjunto de um ou mais dígitos	INT_LIT	o lexema, convertido em `int`, deve ser copiado numa variável C global chamada `VAL_INT`, de tipo `int`.
Um número com ponto flutuante (ver abaixo).	F_LIT	o lexema, convertido em `double`, deve ser copiado numa variável C global chamada `VAL_DOUBLE`, de tipo `double`.

No caso do `F_LIT`, a descrição é a seguinte: qualquer conjunto possivelmente vazio de dígitos, seguido de um ponto ('`.`'), seguido de pelo menos um dígito. Isso tudo pode ser, opcionalmente, seguido de:

1. `e` ou `E`, obrigatoriamente seguido de
2. um sinal + ou − ou nada, obrigatoriamente seguido de
3. pelo menos um dígito.

Exemplos de lexemas reconhecidos: `3.14`, `3.14e+0`, `0.0`, `0.0E+0`, `.0E0`. Exemplos de lexemas não reconhecidos: `0.`, `1e10`, `e+10`, `10.1E`.

1.3 Tratamento de erros lexicais

Em alguns poucos casos, a entrada pode conter caracteres (ou seqüências de caracteres) não reconhecidos por nenhuma expressão regular. Observa-se que algumas seqüências que parecem, objetivamente, erradas ao programador não podem ser capturadas pelo analisador lexical. Considere por exemplo a entrada: `10E`. Apesar de este lexema não condizer com nada do que se espera, o scanner irá retornar um `INT_LIT` (lexema `10`) e logo depois um `IDF` (lexema `E`). Na verdade, essa entrada é lexicamente correta, e o "erro" que o ser humano detecta é *sintático*: caberá apenas à gramática do analisador sintático não autorizar uma sucessão de dois tokens `INT_LIT` e `IDF`.

Isso dito, existem casos de erros lexicais. Por exemplo, se um caractere '?' aparecer na entrada, nenhuma das expressões regulares acima definidas deverá aceitá-lo. Isso acontece também com outros casos. Para pegar tais casos, a solução no Lex é incluir uma última regra (em último lugar para que seja aplicada unicamente em última instância, ou seja se nada fechou antes), do tipo:

```
. { printf(``Erro lexical - caractere nao reconhecido: %c.\n'', yytext[0]);
    exit(-1); }
```

A expressão regular `.` no início da linha significa "qualquer caractere" (único). Ao reconhecer qualquer caractere, então, o analisador lexical gerado pelo Flex irá imprimir na tela uma mensagem relatando o caractere encontrado e não reconhecido, e depois interromper a execução do programa (`exit`) com o código de retorno negativo, associado a um erro. Observa-se que se usa, no `printf`, a variável interna ao Lex chamada `yytext`. Essa variável, que iremos usar mais adiante, é um string (`char*` em C) que contém sistematicamente o lexema que está sendo comparado com a E. R. Como, nessa régra, se usa a expressão regular `.` que autoriza apenas um caractere, `yytext` é aqui um string de tamanho 1, e portanto `yytext[0]` é o único caractere compondo o lexema, que é no caso o caractere que não foi reconhecido. (Para amadores de ponteiros em C, podia se escrever também `*yytext` em lugar de `yytext[0]`.)

O uso de `.` nessa regra deixa claro que se deve aplicá-la apenas como último recurso: é uma regra "pega tudo".

1.4 Observações finais

Essa etapa 1 é simples - basicamente, consiste em preencher arquivos existentes com as Expressões Regulares vistas em aula, de acordo com a sintaxe do Flex. Com os arquivos providos, tudo deve se compilar automaticamente. Posto isso, é muito importante dedicar tempo para *testar* seu analisador lexical e verificar o maior número possível de entradas para verificar que ele retorne os tokens que se espera.

Os testes dessa etapa irão principalmente testar essas expressões regulares.

Chapter 2

Especificação da Etapa 2

Pede-se implementar uma estrutura de árvore generalizada (ver o arquivo `node.h`), e usá-la junto com um analisador sintático (*parser*) a ser obtido pelo Yacc.

A árvore generalizada é uma árvore em que cada nó pode haver qualquer número de filhos. O arquivo `node.h` provê uma implementação do tipo abstrato `Node` (atalho de escrita para `struct _node *`) com as informações que se quer armazenar dentro do nó. Espera-se, entre outros procedimentos, implementações do construtor `create_node`, do acessor ao filho i de um nó, de um destrutor `deep_free_node`, de um método que retorne a altura da árvore cuja raiz seja um dado nó, e de um método que retorne o número total de nós na árvore.

Salienta-se que se espera implementações *corretas*: sem perda de memória (lembrar que C não usa coletor de lixo: caso se use um `malloc`, deve-se usar um `free` em algum outro lugar), sem acesso fora da área de memória alocada, sem efeito colateral não controlado, etc...

O fim deste capítulo, a partir da página 20, detalha o perfil de cada um dos procedimentos exigidos. Cuidado: alguns deles devem levantar erros em casos detalhados na especificação.

O analisador sintático. Deve-se implementar um *parser* (analisador sintático) bottom-up, graças ao Yacc, que reconhece a gramática usada no compilador `Pico` e cria uma árvore que representa as derivações sucessivas (*Abstract Syntax Tree*).

Você *deve*:

- Continuar a usar sua implementação de `Pico` da etapa 1;
- ler a apostila que apresenta como se usa Yacc (no Moodle);
- baixar a gramática Yacc `pico.y` e o arquivo `node.h` disponibilisados no Moodle;
- copiá-los e complementá-los em `Pico/src` para implementar a estrutura de dados de árvore e o parser com a geração da AST.

O trabalho de implementação é mínimo, mas tem trabalho para entender como se faz.

2.1 Descrição da Gramática

Nota-se que a gramática formal é fornecida, na sintaxe do Yacc. Descreve-se a seguir essa gramática, parte por parte. Como de praxe no Yacc, os terminais da gramática (tokens) estão escritos em letras maiúsculas.

Um programa `Pico` é composto por uma seção opcional de declarações, seguida de uma seção de ações. A seguinte regra reconhece essa divisão:

```
code: declaracoes acoes
    | acoes
```

2.1.1 Declarações

A seção de declarações pode não existir, ou é composta por uma ou mais declaração, separada(s) pelo terminal `';'`. Uma declaração é composta por uma lista de identificadores, seguida por `':'`, seguido por um indicador de tipo. Uma lista de identificadores consiste em um número qualquer de (pelo menos um) identificador(es) separado(s) por `','`.

```
declaracoes: declaracao ';'
           | declaracoes declaracao ';'
           ;

declaracao: tipo ':' listadeclaracao

listadeclaracao: IDF
               | IDF ',' listadeclaracao
               ;
```

Um indicador de tipo pode derivar em:

- cada um dos 4 terminais que informa um tipo (INT, DOUBLE, FLOAT ou CHAR, ver o capítulo sobre o scanner, na etapa 1.STRING não será considerado), ou
- cada um dos 4 terminais que informa um tipo, seguido de `'('`, seguido de uma lista de duplas de inteiros, seguida de `')'`.

  Uma lista de duplas de inteiros é composta por qualquer número (maior ou igual a 1) de ocorrências da seqüência de três símbolos seguintes: `INT_LIT ':' INT_LIT`, cada ocorrência sendo separada da seguinte por `','`.

```
tipo: tipounico
    | tipolista
    ;

tipounico: INT
         | DOUBLE
         | REAL
         | CHAR
         ;

tipolista: INT '(' listadupla ')'
         | DOUBLE '(' listadupla ')'
         | REAL '(' listadupla ')'
         | CHAR '(' listadupla ')'
         ;

listadupla: INT_LIT ':' INT_LIT
          | INT_LIT ':' INT_LIT ',' listadupla
          ;
```

Exemplo 1 de declarações:

```
double : x, tmp_1;
```

Exemplo 2 de declarações:

```
real:x;

real  : z1t2     ,     i    ;
```

Exemplo 3 de declarações:

```
real:x;
double(10:20, 30:10, 0:5): tmp;
```

Neste último caso, encontra-se uma lista de três duplas de inteiros: `10:20`, `30:10` e `0:5`.

Comentário. Observa-se que nessa etapa de definição da sintaxe, não se associa semântica ainda a essas construções. No entanto, parece intuitivo que este tipo de declaração servirá para definir um array de 3 dimensões e que cada dupla `n:m` define os valores mínimos e máximos autorizados para o índice em uma direção. Nessa lógica, a segunda dupla `30:10` parece dever levar a um erro, pois se está tentando declarar um array com um limite inferior de índice maior do que o limite superior na segunda dimensão. Repara-se que isso seria um erro semântico, tipicamente detectado na próxima etapa, e não nessa.

2.1.2 Ações

A seção de ações é composta por um ou mais comando(s), cada comando sendo terminado pelo terminal `';'`. Um comando pode ser composto por:

- uma atribuição, do tipo: `lvalue '=' expr`, ou
- um enunciado simples.

`lvalue` pode ser ou um identificador simples, ou um identificador seguido por `'['`, seguido por uma lista de `expr`, seguida por um terminal `']'` final. Uma lista de `expr` é composta por pelo menos uma `expr`; se tiver mais de uma, devem ser separadas por `','`.

```
acoes: comando ';'
    | comando ';' acoes
    ;

comando: lvalue '=' expr
       | enunciado
       ;

lvalue: IDF
      | IDF '[' listaexpr ']'
      ;

listaexpr: expr
         | expr ',' listaexpr
         ;
```

Uma `expr` abrange:

- duas `expr` separadas por um dos terminais que representa um operador aritmético; ou
- uma `expr` entre parênteses; ou
- um literal, inteiro ou de ponto flutuante; ou
- uma `lvalue` tal como definida acima; ou
- uma chamada a um procedimento.

  Uma chamada a procedimento consiste em um identificador, seguido de uma lista de `expr` entre parênteses.

```
expr: expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '(' expr ')'
    | INT_LIT
    | F_LIT
    | lvalue
    | chamaproc
    ;

chamaproc: IDF '(' listaexpr ')'
         ;
```

Um enunciado simples se limita a uma `expr` ou a uma instrução de controle. Uma instrução de controle pode ser:

- `IF`, seguido de `'('`, seguido de uma expressão booleana, seguida de `')'`, seguido de `THEN`, seguido de um ou vários comando(s) (no sentido definido acima), seguido de:
  - Ou o terminal `END`;
  - Ou o terminal `ELSE`, seguido por um ou vários comandos (no sentido definido acima), seguido(s) pelo terminal `END`.
- `WHILE`, seguido de `'('`, seguido de uma expressão booleana, seguida de `')'`, seguido de `'{'`, seguido por um ou vários comandos (no sentido definido acima), seguido(s) por `'}'`.

```
enunciado: expr
         | IF '(' expbool ')' THEN acoes fiminstcontrole
         | WHILE '(' expbool ')' '{' acoes '}'
         ;

fiminstcontrole: END
               | ELSE acoes END
               ;
```

Por fim, uma expressão booleana é definida recursivamente da forma seguinte: pode ser:

- Ou o terminal `TRUE`;

- Ou o terminal `FALSE`;
- Ou uma expressão booleana entre parênteses;
- Ou duas expressões booleanas separadas por um dos operadores lógicos reconhecidos pelo scanner (`AND`, `OR`);
- Ou uma expressão booleana precedida pelo token `NOT`;
- Ou duas `expr` separadas por um dos operadores relacionais (tokens: `'<'`, `'>'`, `LE`, `GE` e `EQ`).

```
expbool: TRUE | FALSE | '(' expbool ')' | expbool AND expbool
       | expbool OR expbool | NOT expbool | expr '>' expr
       | expr '<' expr | expr LE expr | expr GE expr | expr EQ expr
       | expr NE expr
       ;
```

Exemplos de ações:

```
x = 1;
f(2-3,x) + tmp1;
if ( g(tmp[1,2]*2) <= 10 ) then
   while ( (x>0) & y!=1  ) { x=0; } ;
end
```

Exemplo de programa `Pico` inteiro:

```
int: i, n ;
double: x ;
x = 2.0; i=0 ; n=10;
while (i <= n) {
   x = x*2 ; i = i+1;
};
```

Nota-se o uso de ponto-virgula depois de um enunciado do tipo `while` (ou `if`...): sendo enunciados, a gramática os obriga a serem seguidos do terminal `;`.

2.2 Geração da Árvore de Sintaxe Abstrata (AST)

A árvore sintática é uma versão simplificada da árvore de derivações. Elimina-se todas as derivações diretas do tipo $X \longrightarrow Y$ onde X e Y são dois símbolos não-terminais quaisquer. A sub-seção a seguir explica em nível teórico o que fazer; a sub-seção 2.2.2 detalha a forma como se implementa isso em C com o Yacc.

2.2.1 Uso das Reduções e Geração da AST

Ao reduzir uma seqüência de símbolos $X_1 X_2 \ldots X_n$ em X (pela produção $X \longrightarrow X_1 X_2 \ldots X_n$), o Yacc deverá criar um novo nó na árvore, constituído:

- de um novo rótulo (identificador único do nó);
- do tipo apropriado, definido em função do não-terminal X sendo reduzido (por exemplo: `expr`, `decl`, etc...). Lembra-se que o arquivo `node.h` contém exemplos de definição de constantes que servem para definir esses tipos. A lista pode ser alterada e completada;

- dos nós filhos, que serão os nós associados aos símbolos $X_i, i = 1 \ldots n$.

Obviamente, deverá ser usada a estrutura de dados definida em `node.h` para fazer isso. A árvore total deverá ser acessível, ao se encerrar a análise sintática, através de uma variável global chamada `syntax_tree` e de tipo `Node*`. Isso será feito tipicamente pela ação associada à última redução, ou seja à redução associada à produção a partir do símbolo start da gramática.

Quando um símbolo X_i é um terminal (token), por definição o nó associado é uma folha. Neste caso, e apenas neste caso, a folha a ser criada deverá também informar o lexema associado ao token reconhecido. Para recuperar o lexema, usa-se a variável `yytext`.

Quando existe apenas um X_i derivando a partir de X, não se precisa criar um novo nó na árvore: apenas se aponta, por X, o nó apontado por X_i.

Um teste importante a considerar é a "decompilação-compilação" de programas `Pico`. Seja o programa fonte `foo0.pico`. Ao ser compilado, ele gera a árvore `syntax_tree`. Pode-se decompilar essa representação abstrata de `foo0.pico`, percorrendo-se a árvore para imprimir (de esquerda para direita) cada lexema contido nos nós folha. Escrevendo-se essa seqüência de lexemas em um arquivo `foo1.pico`, deve-se recuperar um programa semanticamente eqüivalente ao original (a única diferença sendo os espaços brancos). Assim, se se compilar `foo1.pico`, deve-se obter uma segunda árvore sintática idêntica à primeira. O arquivo `node.h` especifica um procedimento chamado `uncompile()` que tem essa funcionalidade.

2.2.2 Ações em C no Yacc

Cada regra de produção do tipo:

```
symbol : xx yy zz
```

no Yacc pode ser seguida de um trecho de código em C, da mesma forma como no Lex:

```
symbol : xx yy zz   { x = run_whatever(); printf("%d",x); }
```

seria um exemplo.

Nota-se que os símbolos `xx`, `yy` e `zz` podem ser terminais ou não-terminais, e que pode haver tantos símbolos como se quer.

A chave é entender que esse trecho de código será executado, durante a fase de análise sintática, ao aplicar a redução `xx yy zz` em `symbol`.

2.2.2.1 Ações e Atributos

É muito comum que este código seja usado para calcular os valores dos atributos associados aos símbolos envolvidos (por exemplo: o valor do lexema associado a um `INT_NUM`). Você vai usar tais atributos abstratos para gerar a árvore de sintaxe abstrata (AST). Esses atributos são referenciados, no Yacc, pelo uso das variáveis `$1`, `$2`, ..., `$n`, sendo `$i=1,..,n` o atributo do i-ésimo símbolo à direita da produção.

Por exemplo:

```
symbol : xx yy zz   { x = run_whatever($2); printf("%d",x); }
```

calcularia o valor de `x` a partir do atributo do símbolo `yy` (que é o segundo que aparece à direita da produção).

O atributo associado ao símbolo à esquerda (alvo da redução), `symbol` no exemplo, é `$$`. Este atributo é bastante importante para inicialisar o cálculo dos atributos, ao reduzir um único símbolo terminal em um não-terminal. Por exemplo:

```
expr : INT_LIT   { $$ = atoi(yytext); }
```

expressa que o atributo do não-terminal `expr` deve ser inicializado a partir do valor numérico do lexema que foi reconhecido pelo analisador lexical como sendo associado a um token `INT_LIT`. Na verdade, essa forma de escrever é um atalho que não irá funcionar, pois a variável `yytext`, usada no Lex, não é acessível pelo Yacc.

A forma correta é passar por uma variável global, conhecida por ambos o Lex e o Yacc, chamada `yylval`. Essa variável é automaticamente usada pelo Yacc para informar o valor do atributo associado a um token[1]. Assim, do lado do lex, se escreve:

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

(A regexp acima é simplificada, obviamente.) E do lado do Yacc, se pode agora escrever:

```
expr : INT_LIT   { $$ = $1; }
```

2.2.2.2 Tipo dos Atributos

Nos exemplos acima, o atributo é de tipo inteiro. É o *default* no Yacc. No entanto, pode-se querer atributos de qualquer tipo; nessa etapa, deve-se usar atributos de tipo `Node*` ou `char*` (para recuperar os lexemas). Para isso, deve-se re-definir o tipo dos atributos associados aos símbolos e aos tokens. Por exemplo, para os símbolos `expr` e `INT_LIT`, se faz isso da forma seguinte (no cabeçalho do arquivo .y):

```
%union {
  Node* noh;
  char* cadeia;
}

%type< noh > expr
%token< cadeia > INT_LIT
```

Pode-se depois usar os atributos através das variáveis `$$`, `$1`, etc... no Yacc, e de `yylval` do lado do Lex, por exemplo através de:

```
[0-9]*   { yylval.cadeia=(char*)malloc((strlen(yytext)+1)*sizeof(char));
           strcpy(yylval.cadeia, yytext); return(INT_LIT); }
```

2.3 Trabalho a ser entregue

Seu parser chamado `pico` deve aceitar como argumento na linha de comando o nome do arquivo contendo o input (já era o comportamento do seu scanner). O arquivo de entrada deve obrigatoriamente ser sufixado pela extensão `.pico`.

A saída de `Pico`, na tela, deve ser a seguinte:

[1] Na verdade, a variável $1 é um ponteiro sobre `yylval`.

- `OKAY` no caso que a entrada esteja de acordo com a gramática;
- `ERROR` no caso contrário.

Ademais, ao sair do laço while que efetua a análise sintática, a variável global `syntax_tree` deve apontar para a raiz da árvore sintatica tal como definida nas seções anteriores.

Deve-se alterar o código do analisador lexical para que não seja mais impresso na tela a palavra `SPACE` ao reconhecer um espaço branco na entrada.

Para compilar seus analisadores lexicais e sintáticos, e obter um executável, um Makefile é provido. Nota-se que a partir dessa etapa 2, o `main()` do compilador se encontra em um arquivo separado da entrada do Flex (`scanner.l`) e da entrada do Yacc (`pico.y`). É uma das formas de organizar as várias componentes do código fonte, e o Makefile provido reflete isso. Tem-se liberdade para alterar o `main()`.

2.4 Procedimentos relativos à Árvore

O arquivo `node.h` especifica os procedimentos que devem ser implementados para criar e acessar os nós da árvore generalizada que implementa a AST. A árvore é independente do *parser* e do Yacc. Em particular, a árvore deve prever que um dado nó pode ter tantos filhos como for necessário para representar uma árvore de derivação, dada uma gramática. Para não limitar *a priori* o número de filhos de um dado nó, pede-se usar uma lista (simplesmente) encadeada para gerenciar os filhos de um dado nó.

As seções que seguem foram geradas diretamente a partir dos comentários incluídos nos arquivos fontes. Pode-se ler os mesmos diretamente, além da versão HTML que se encontra em `Pico/src/html/`.

2.5 Referência do Arquivo node.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
```

Estruturas de Dados

- struct _node

Definições e Macros

- #define **code_node** 298
- #define **declaracoes_node** 299
- #define **declaracao_node** 300
- #define **tipo_node** 301
- #define **int_node** 501

Definições de Tipos

- typedef int **Node_type**
- typedef struct _node **Node**

Funções

- Node ∗ create_node (int nl, Node_type t, char ∗lexeme, Node ∗child0,...)
- int nb_of_children (Node ∗n)
- int is_leaf (Node ∗n)
- Node ∗ child (Node ∗n, int i)
- int deep_free_node (Node ∗n)
- int height (Node ∗n)
- void uncompile (FILE ∗outfile, Node ∗n)

Variáveis

- Node ∗ **syntax_tree**

2.5.1 Descrição Detalhada

Versão:

1.1

2.5.2 Funções

2.5.2.1 Node* child (Node * *n*, int *i*)

accessor to the i'th child of a Node.

Parâmetros:

n : the node to be consulted. Must abort the program if 'n' is NULL.

i : the number of the child that one wants. Must be lower than the degree of the node and larger than 0. Must abort the program if i is not correct.

Retorna:

a pointer on a Node.

2.5.2.2 Node* create_node (int *nl*, Node_type *t*, char * *lexeme*, Node * *child0*, ...)

* Node constructor.

Parâmetros:

nl,: line number where this token was found in the source code.

t,: node type (one of the values define'd above). Must abort the program if the type is not correct.

lexeme,: whatever string you want associated to this node.

attr,: a semantical attribute.

child0,: first of a list of pointers to children Node*'s. See the extra file 'exemplo_func_var_arg.c' for an example. To create a leaf, use NULL as last argument to create_node().

Retorna:

a pointer to a new Node.

2.5.2.3 int deep_free_node (Node * *n*)

Destructor of a Node. Desallocates (recursively) all the tree rooted at 'n'.

2.5.2.4 int height (Node * *n*)

returns the height of the tree rooted by 'n'. The height of a leaf is 1.

2.5.2.5 int is_leaf (Node * *n*)

Tests if a Node is a leaf. Must abort the program if 'n' is NULL.

Retorna:

1 if n is a leaf, 0 else.

2.5.2.6 int nb_of_children (Node * *n*)

accessor to the number of children of a Node. Must abort the program if 'n' is NULL.

2.5.2.7 void uncompile (FILE * *outfile*, Node * *n*)

Prints into a file the lexemes contained in the node rooted by 'n'. The impression must follow a depth-first order.

Parâmetros:

outfile : the file to which the lexemes are printed.

n : the root node of the tree. Must abort the program if 'n' is NULL.

Chapter 3

Especificação da Etapa 3

A etapa 3 consiste em tratar:

- a inserção na tabela de símbolos de variáveis declaradas;
- a geração de código de três endereços ("TAC") das expressões aritméticas, inteiras e com ponto flutuante.

Para o primeiro ponto, provê-se uma implementação de uma tabela de símbolos (ver o código entregue `src/symbol_table.[ch]`). Para o segundo, deve-se implementar os atributos semânticos apropriados (tais como vistos em aula: `codigo`, `local`,...) para os símbolos da gramática envolvidos (`expr`,...) e usar as ações semânticas em C no arquivo Yacc da segunda etapa para completar a criação dos nós com o cálculo apropriado de seus atributos semânticos.

Resolução de parte dos conflitos na gramática. O Yacc possibilita indicar a precedência e a associatividade de operadores. Por exemplo, pode-se indicar que o parsing de `2+x+3` deve ser efetuado na ordem `(2+x)+3` (`+` é dito associativo à esquerda), e que `2+x*3` deve ser analizado como `2+(x*3)` (`*` possui precedência maior do que `+`). Essas escolhas possibilitam tirar vários conflitos Shift-reduce que podem ter acontecido em suas gramáticas.

Para isso, basta inserir as linhas abaixo na primeira parte do arquivo pico.y (por exemplo depois das definição dos tokens):

```
%left OR
%left AND
%left NOT
%left '+' '-'
%left '*' '/'
```

A ordem das linhas define a precedência crescente dos operadores. Os `left` e `right` definem a associatividade de cada um.

(Em linguagens que possibilitam múltiplas atribuições (`x = y = 1`), é comum incluir uma linha `%right '='` para reduzir na ordem `x = (y=1)`. `Pico` não autoriza isso.)

Inserção de nova produção na gramática. Com a finalidade de se obter uma visualização e depuração do código final, é necessária alguma forma de emitir saídas na tela. Para alcançar esse objetivo, deverá ser incluído um novo comando 'print' na linguagem `pico`, que ofereça tal funcionalidade. Primeiramente,

deve-se definir um novo token PRINTF no arquivo tokens.h. Basta incluir um novo #define neste arquivo. Além disso, o arquivo scanner.l deverá adicionar a seguinte expressão regular para tratar o token PRINTF:

```
"print" { return(PRINTF); }
```

Por último, deverá ser acrescentada uma nova produção na gramática pico.y:

```
enunciado: PRINTF '(' expr ')'
```

Deve ser criada uma ação semântica relativa a criação dos nodos dessa produção na árvore sintática, semelhante ao realizado para as demais produções na etapa 2.

3.1 Declaração de variáveis

3.1.1 Tabela de Símbolos

Para gerenciar os identificadores na linguagem Pico, provê-se uma estrutura de dados para a tabela de símbolos. A tabela de símbolos é implementada através de uma tabela de Hash. Uma entrada na tabela de símbolos será caracterizada por vários campos (ver o tipo abstrato entry_t em symbol_table.h), incluindo um campo "nome" de tipo char* (ou string, em C). A partir deste nome, calcula-se um hash para associá-lo a um número inteiro, que irá servir para acessar um vetor. Se houver colisões na função de Hash, cuidados são tomados para desempatar. As funções suportadas são de inserção de uma entrada, de consulta, além de funcionalidades de verificação e de impressão do conteúdo de uma tabela de símbolos.

O perfil exato de cada um dos procedimentos definidos para a tabela de símbolos se encontra no fim deste capítulo, p. 31.

3.1.2 Ações semânticas

Ao efetuar o *parsing* das declarações, deve-se identificar o tipo abstrato de cada variável e inserir na tabela de símbolos, para cada uma delas, uma entrada que represente de forma abstrata:

- O lexema associado ao token IDF sendo analizado;
- O tipo da variável;
- O espaço na memória que se deverá usar para armazenar seu valor;
- O seu endereço, ou seja o deslocamento a partir do endereço base aonde se encontrará mapeada a variável;
- Informações extras no caso de um *array* (ler mais adiante).

No Pico-2011, tem-se apenas um escopo único e global para todas as variáveis, e por isso se usará apenas uma tabela de símbolos global chamada s_table. *Todas as variáveis devem ter sido declaradas antes de serem usadas no programa Pico. Uma variável referenciada sem ter sido declarada previamente deverá levar seu compilador a disparar um erro* UNDEFINED_SYMBOL

Um erro UNDEFINED SYMBOL SERIA provocado, em C/Yacc, da forma seguinte:

```
#define UNDEFINED_SYMBOL_ERROR -21
/* muitas linhas de codigo, pois o define acima esta lah no topo do pico.y */
```

```
/* ... */
XXX: YYY {
    printf(``UNDEFINED SYMBOL. A variavel %s nao foi declarada.\n'', lexema);
    return( UNDEFINED_SYMBOL_ERROR );
    }
```

(Onde `XXX`, `YYY` e o string `lexema` estão usados de forma genérica, obviamente. Cabe-lhes identificar as ações onde se deve usar tal disparo de erro.)

Os tipos básicos terão o tamanho seguinte:

- tipo char: 1 Byte;
- tipo int: 4 Bytes;
- tipo real: 4 Bytes;
- tipo double: 8 Bytes.

O deslocamento e tamanho da variável deverão ser calculados à medida que o *parsing* anda, conforme explicado em aula.

Arrays. No caso dos arrays, o trabalho de representação do tipo pode ser mais complexo. No `Pico`, iremos nos limitar ao caso de arrays de tipos simples. A representação de arrays (de arrays)* de arrays[1] necessita o uso de um grafo abstrato que não vai ser implementado neste semestre. Para arrays de tipos simples, deverá ser reservado o espaço na memória igual ao número total de elementos no array, multiplicado pelo tamanho do tipo simples (em Bytes). Na tabela de símbolos, uma entrada associada a um array deverá, obviamente, informar este tamanho, e também conter os valores pre-calculados dos elementos necessários ao cálculo semântico do acesso a um elemento indexado no array (ver a aula sobre arrays multidimensionais).

Assim, a `struct entry_t` entregue nesta Etapa 3, que continha um campo `void* extra`, deverá ser usada e complementada para que este campo se torne um ponteiro para uma nova struct, com todas as informações necessárias (por exemplo: a constante "c"; o número de dimensões; os limites inferiores e superiores em cada uma das dimensões; etc...). Caberá à implementação consultar esses campos, quando há necessidade numa ação semântica.

Observa-se que vocês têm uma certa margem de manobra nessa representação interna de seus tipos, em especial no caso dos arrays.

Ao terminar a análise sintática da declaração de variáveis, o tamanho total usado na memória para todas as variáveis deverá ser armazenado.

3.2 Código TAC a ser gerado

Uma instrução TAC deverá ter o formato seguinte:

```
z := x OP y
```

onde `x`, `y` e `z` deverão ser endereços na memória e `OP` um operador especificado abaixo. Em alguns casos, um dos operandos `x` ou `y` pode estar vazio.

[1] Para quem não entendeu o asterisco, isso era uma expressão regular...

Endereços na memória. Serão representados através de um deslocamento a partir de um endereço base, tipicamente o endereço base de um segmento de pilha. Supor-se-á que esta base se encontra num registrador `SP`, e o endereço na memória será escrito `d(SP)`. Por exemplo, uma variável que for mapeada para o endereço 16 (Bytes) terá como endereço `016(SP)`.

Um tratamento diferente será dado às variáveis temporárias, as quais deverão ser mapeadas através de um deslocamento relativo a um registrador diferente, chamado por convenção `Rx`.

Operadores TAC. Usar-se-á a sintaxe seguinte para os operadores da linguagem TAC:

1. Operações aritméticas com números inteiros se chamarão: `ADD`, `SUB`, `MUL` e `DIV`. Todas terão dois operandos.
2. Operações aritméticas com números em ponto flutuante se chamarão: `FADD`, `FSUB`, `FMUL` e `FDIV`. Todas terão dois operandos.
3. O símbolo `:=` será usado entre o operando `z` e os dois operandos `x` e `y`, a semântica dessa atribuição sendo que os endereços aparecendo à direita serão de-refenciados (usar-se-á o conteúdo dos endereços das variáveis), e que o resultado estará armazenado no endereço encontrado à esquerda. Valores numéricos aparecendo à direita serão considerados como valores imediatos.

   Assim, `z := z+1` significa: consulta a memória no endereço de `z`, soma o valor nele armazenado com 1, e armazena o valor resultante no endereço de `z` (ou seja, curto e grosso: faz o que se espera...).
4. `PRINT`, com um operando inteiro único (sintaxe: `PRINT x`)

O código TAC inteiro será composto por:

1. um "cabeçote" que informará dois números inteiros, cada um numa linha separada, que definirão repectivamente o espaço total (em Bytes) na memória necessário para armazenar as variáveis manipuladas pelo programa, e (na segunda linha) os temporários.
2. a seqüência de instruções atômicas, cada qual possuindo um número de linha separado da instrução por `:`. O número deverá ser formatado para caber em exatamente 3 caracteres, sendo os números menores do que 100 completados à esquerda com zeros (ou seja, irá de `000` até `999`). O `:` deverá ser seguido por 3 caracteres "espaço branco". Os deslocamentos deverão ser formatados para caber em exatamente três caracteres, também sendo completados à esquerda com zeros se for o caso.

   Para formatar, em C, essas instruções, recomenda-se usar a função `sprintf`. Observa-se que essas cobranças relativas à formatação precisam ser cumpridas na hora de imprimir uma listagem do código TAC, e não obrigatoriamente deve se refletir na representação interna do código.

Exemplo de código TAC esperado:

```
88
8
000:   084(SP) := 2
001:   000(Rx) := 084(SP) MUL 5
002:   004(Rx) := 000(Rx) SUB 3
003:   000(SP) := 004(Rx)
004:   PRINT 000(SP)
```

(Esse trecho de código poderia representar a compilação de
`int: x; real: tab(1:20); int: y; y = 2; x = y*5 - 3; print(x)`, onde `A` seria um array bidimensional de inteiros, tendo 10 elementos na dimensão 1 e 20 na dimensão dois (por isso

necessitando 10×20× 4 Bytes), onde `z` estaria mapeado no endereço 0, `i` no endereço 8, `j` no endereço 12 e `A` teria seu endereço base no endereço 16.)

Nota-se que essa linguagem TAC é de nível mais baixo do que a linguagem usada nas aulas. A fins de depuração, aconselha-se programar primeiramente ações semânticas que geram código TAC com variáveis "simbólicas" (em particular TMP0, TMP1, etc...). Pode-se usar essa primeira versão e a tabela de símbolos, para gerar o código TAC final numa segunda fase. Opcionalmente, pode-se programar um flag `-v` do `Pico` para controlar seu output na forma "TAC alto nível" ou TAC baixo nível. Apenas será exigido o TAC "baixo-nível".

Código auxiliar para geração de código TAC Será necessário implementar pelo menos:

1. Uma representação interna de uma instrução TAC. Por exemplo, pode-se usar uma struct de quatro membros para isso, tal como foi entregue no `lista.h`.

2. Uma lista encadeada de instruções TAC.

   A lista encadeada deverá suportar as operações de inserção (no fim da lista), de concatenação de duas listas e de impressão do seu conteúdo. Observa-se que a função de impressão pede por prefixar cada elemento da lista por um número, num formato especificado, que indica o número do elemento na lista.

3. Um procedimento para gerar "novos temporários".

4. Um procedimento para gerar "novos rótulos".

Nada está imposto sobre a implementação interna dessas estruturas de dados e procedimentos, porém será considerada a qualidade da implementação na avaliação. Aconselha-se fortemente testar de forma unitária essa lista. A lista deve respeitar os perfis de procedimentos (ver também `src/lista.h` no código fornecido) definidos p. 34.

3.3 Ações semânticas a serem implementadas

3.3.1 Expressões aritméticas

Todas as ações semânticas necessárias à geração de código para expressões aritméticas devem ser implementadas. Entende-se, nessa etapa, por “expressão aritmética”, todas as derivações possíveis, tais como definidas na Etapa 2, a partir de uma atribuição do tipo `lvalue '=' expr` (ver Sec. 2.1.2, p. 15), SENDO EXCLUÍDAS as derivações de `expr` em chamada a um procedimento. Informalmente, isso significa que se espera qualquer combinação de operadores aritméticos atuando em identificadores ou arrays, os quais podem aparecer tanto à esquerda como à direita de um sinal de atribuição.

O emprego de números em ponto flutuante **não** será cobrado nessa etapa, tratando apenas operações com números inteiros.

O fato de incluir os identificadores nas expressões, assim como os arrays, significa que se deverá efetuar as ações semânticas relevantes de consulta à tabela de símbolos, ao encontrar um `IDF`, para verificar se foi declarado, e se sim recuperar as informações necessárias à geração de código (se é relevante).

No caso de uma expressão derivar em um terminal `INT_LIT`*, deve-se usar o próprio lexema na instrução TAC gerada. Por exemplo, ao compilar* `z = y + 314`*, o código TAC gerado irá conter uma linha com* `314` *explicitamente escrito (numa instrução ADD).*

3.3.2 Instrução print

Deve-se gerar código TAC, através de ações semânticas, para o único caso de chamada a procedimento que é a chamada a `print`.

3.4 Trabalho a ser entregue

Seu parser chamado `pico` deve aceitar como argumento na linha de comando:

- o nome do arquivo contendo o input (já era o comportamento do seu scanner). O arquivo de entrada deve obrigatoriamente ser sufixado pela extensão `.pico`. Este argumento deve ser o último da linha de comando.
- o flag `-o`, seguido pelo nome do arquivo de saída. Este nome deve ser sufixado pela extensão `.tac`. O código TAC gerado pelo compilador deve ser escrito neste arquivo (se a compilação for bem sucedida).

Exemplos de uso:

```
pico -o test1.tac test1.pico
pico -o output.tac input.pico
pico -o resultado.tac prog_1_.pico
```

Para testar se seu código TAC é correto, poderá ser usado o script que converte um programa TAC para assembly x86 (tradutor), compilar o mesmo e executá-lo. A seção seguinte explica como funciona.

3.5 Testes e tradutor TAC para x86

Para testar essa etapa, você poderá converter o código TAC em assembly x86. A conversão de código TAC em *assembly* se faz como segue.

3.5.1 Pré-Requisitos

Para transformar o programa gerado em código de máquina, é necessário um ambiente Linux com os seguintes programas instalados e funcionais:

1. GNU Assembler ("`as`"),
2. GNU Linker ("`ld`"),
3. GNU libc ("`glibc`"),
4. Um interpretador Python.

Em geral, a maioria das distros Linux (*e.g.*, Ubuntu) vêm com esses programas instalados.

Fornecemos um script Python tradutor que, dado um código TAC gera o correspondente *assembly* x86. Com ele, pode-se fazer o seguinte:

1. Chamar o tradutor em cima do arquivo passado como parâmetro, que transforma o código TAC em um código *assembly*. O arquivo resultante deve se chamar "output.s"

2. Chamar o GNU Assembler sobre o arquivo "output.s", resultando no arquivo "output.o", através do comando seguinte:

```
as output.s -o output.o
```

3. Chamar o GNU Linker sobre o arquivo "output.o", gerando como saída o arquivo "output", que é um executável válido. Deve-se ligar o código à `glibc`, pra que se possa usar a função `printf` dentro do código assembly. Isso se faz através do comando:

```
ld -dynamic-linker /lib/ld-linux.so.2 -o output -lc output.o
```

(O diretório `/lib/ld-linux.so.2` é a localização da `glibc` para o Ubuntu, podendo variar para outras distribuições. Consulte a documentação própria.)

3.5.2 Implementação do Tradutor

O tradutor é um analisador de expressões regulares que irá ler uma linha de instrução TAC (no formato descrito nas etapas anteriores) e transformá-la em algumas linhas *assembly* x86. Não é necessário um Compilador (análise sintática e semântica), pois a especificação do TAC foi feita em cima de uma Linguagem Regular, propositalmente.

O tradutor usa um arquivo ("model.template") para gerar o assembly, que é fornecido. Esse arquivo já está pronto para ser processado pelo `as`, faltando apenas preencher automaticamente as seguintes lacunas:

1. `<stack_size>` : o tamanho da pilha. É o número da primeira linha do código TAC.
2. `<temp_size>` : tamanho reservado às variáveis temporárias. É o número na segunda linha do código TAC.
3. `<code>` : é o código *assembly* equivalent ao código TAC.

O template é copiado para o arquivo "output.s", substituindo nele as marcações `<stack_size>`, `<temp_size>` e `<code>` com os dados requeridos, obtidos pelo tradutor ao efetuar a análise do TAC.

Geralmente, uma instrução TAC é convertida para $n > 1$ instruções assembly, pois há necessidade de salvar registradores, usá-los e restaurar seu conteúdo. *e.g.*,

```
003:   012(Rx) := 008(Rx) ADD 016(SP)
```

é convertida em

```
_003: MOVL  8(%EDX) , %EAX
      ADDL  16(%ECX) , %EAX
      MOVL  %EAX ,12(%EDX)
```

A instrução TAC `PRINT` é implementada através de uma chamada ao procedimento `printf` da linguagem C da seguinte maneira:

```
005:   PRINT 004(SP)
```

é convertida em

```
_005: PUSHL  4(%ECX)
      PUSHL  $intf
      CALL printf
```

3.6 Referência do Arquivo symbol_table.h

```
#include <stdio.h>
```

Estruturas de Dados

- struct entry_t

Funções

- int init_table (symbol_t *table)
  Inicializar a tabela de Hash.
- void free_table (symbol_t *table)
  Destruir a tabela de Hash.
- entry_t * lookup (symbol_t table, char *name)
  Retornar um ponteiro sobre a entrada associada a 'name'.
- int insert (symbol_t *table, entry_t *entry)
  Inserir uma entrada em uma tabela.
- int print_table (symbol_t table)
  Imprimir o conteudo de uma tabela.
- int print_file_table (FILE *out, symbol_t table)
  Imprimir o conteudo de uma tabela em um arquivo.

Variáveis

- typedef symbol_t
  Encapsulacao de um tipo abstrato que se chamara 'symbol_t'.

3.6.1 Descrição Detalhada

Versão:

1.1

3.6.2 Funções

3.6.2.1 void free_table (symbol_t * *table*)

Destruir a tabela de Hash.

'free_table' eh o destrutor da estrutura de dados. Deve ser chamado pelo usuario no fim de seu uso de uma tabela de simbolos.

Parâmetros:

table,uma referencia sobre uma tabela de simbolos.

3.6.2.2 int init_table (symbol_t * *table*)

Inicializar a tabela de Hash.

Parâmetros:

table,uma referencia sobre uma tabela de simbolos.

Retorna:

o valor 0 se deu certo.

3.6.2.3 int insert (symbol_t * *table*, entry_t * *entry*)

Inserir uma entrada em uma tabela.

Parâmetros:

table,uma tabela de simbolos.

entry,uma entrada.

Retorna:

um numero negativo se nao se conseguiu efetuar a insercao, zero se deu certo.

3.6.2.4 entry_t* lookup (symbol_t *table*, char * *name*)

Retornar um ponteiro sobre a entrada associada a 'name'.

Essa funcao consulta a tabela de simbolos para verificar se se encontra nela uma entrada associada a um char* (string) fornecido em entrada. Para a implementacao, e necessario usar uma funcao que mapeia um char* a um numero inteiro. Aconselha-se, por exemplo, consultar o livro do dragao (Aho/Sethi/Ulman), Fig. 7.35 e a funcao HPJW.

Parâmetros:

table,uma tabela de simbolos.

name,um char* (string).

Retorna:

um ponteiro sobre a entrada associada a 'name', ou NULL se 'name' nao se encontrou na tabela.

3.6.2.5 int print_file_table (FILE * *out*, symbol_t *table*)

Imprimir o conteudo de uma tabela em um arquivo.

A formatacao exata e deixada a carga do programador. Deve-se listar todas as entradas contidas na tabela atraves de seu nome (char*). Deve retornar o numero de entradas na tabela. A saida deve ser dirigida para um arquivo, cujo descritor e passado em parametro.

Parâmetros:

out,um descrito de arquivo (FILE∗).

table,uma tabela de simbolos.

Retorna:

o numero de entradas na tabela.

3.6.2.6 int print_table (symbol_t *table*)

Imprimir o conteudo de uma tabela.

A formatacao exata e deixada a carga do programador. Deve-se listar todas as entradas contidas na tabela atraves de seu nome (char∗). Deve retornar o numero de entradas na tabela.

Parâmetros:

table,uma tabela de simbolos.

Retorna:

o numero de entradas na tabela.

3.6.3 Variáveis

3.6.3.1 typedef symbol_t

Encapsulacao de um tipo abstrato que se chama 'symbol_t'.

3.7 Referência do Arquivo lista.h

```
#include <stdio.h>
```

Estruturas de Dados

- struct tac
- struct node_tac

Funções

- struct tac ∗ create_inst_tac (const char ∗res, const char ∗arg1, const char ∗op, const char ∗arg2)

  Construtor de Instrucao TAC.

- void print_inst_tac (FILE ∗out, struct tac i)

  Funcao que imprime o conteudo de uma instrucao TAC.

- void print_tac (FILE ∗out, struct node_tac ∗code)

  Imprime no arquivo apontado por 'out' o conteudo da lista apontada por 'code'.

- void append_inst_tac (struct node_tac ∗∗code, struct tac ∗inst)
- void cat_tac (struct node_tac ∗∗code_a, struct node_tac ∗∗code_b)

3.7.1 Descrição Detalhada

3.7.2 Funções

3.7.2.1 void append_inst_tac (struct node_tac ∗∗ *code*, struct tac ∗ *inst*)

Insere no fim da lista 'code' o elemento 'inst'.

Parâmetros:

code lista (possivelmente vazia) inicial, em entrada. Na saida, contem a mesma lista, com mais um elemento inserido no fim da lista.

3.7.2.2 void cat_tac (struct node_tac ∗∗ *code_a*, struct node_tac ∗∗ *code_b*)

Concatena a lista 'code_a' com a lista 'code_b'.

Parâmetros:

code_a lista (possivelmente vazia) inicial, em entrada. Na saida, contem a mesma lista concatenada com 'code_b'.

code_b a lista concatenada com 'code_a'.

3.7.2.3 struct tac∗ create_inst_tac (const char ∗ *res*, const char ∗ *arg1*, const char ∗ *op*, const char ∗ *arg2*)

Construtor de Instrucao TAC.

Para testes, pode-se usar qualquer string em argumentos.

Parâmetros:

res um char∗.

arg1 um char∗.

op um char∗.

arg2 um char∗. @ return um ponteiro sobre uma 'struct tac'.

3.7.2.4 void print_inst_tac (FILE ∗ *out*, struct tac *i*)

Funcao que imprime o conteudo de uma instrucao TAC.

Parâmetros:

out um ponteiro sobre um arquivo (aberto) aonde ira ser escrita a instrucao.

i a instrucao a ser impressa.

3.7.2.5 void print_tac (FILE ∗ *out*, struct node_tac ∗ *code*)

Imprime no arquivo apontado por 'out' o conteudo da lista apontada por 'code'.

Parâmetros:

out um ponteiro sobre um arquivo (aberto) aonde ira ser escrita a lista (uma linha por elemento).

code o ponteiro para a lista a ser impressa.

Chapter 4

Especificação da Etapa 4

Deve-se gerar código TAC para tratar dos acessos a *arrays* multi-dimensionais. Apenas o código TAC gerado por seu compilador será testado, o que lhes deixa bastante flexibilidade para programar as funcionalidades internas de seu compilador `Pico`.

A página 27 do capítulo anterior já mencionava a necessidade de prever uma representação interna dos arrays, para que as ações semânticas associadas à parte de declaração de arrays pudessem informar na tabela de símbolos as caraterísticas necessárias na geração de código associada aos acessos aos elementos do array. Essas informações, conforme foi explicado em aula, podem incluir o número de dimensões de um array, os valores inferiores e superiores lícitos para indexar o array em cada dimensão, ou qualquer informação necessária a critério do grupo.

Deve-se portanto reconsiderar as produções da gramática de `Pico` que dizem respeito à declaração de variáveis de tipo array:

```
tipolista: INT '(' listadupla ')'
         | DOUBLE '(' listadupla ')'
         | REAL '(' listadupla ')'
         | CHAR '(' listadupla ')'
         ;

listadupla: INT_LIT ':' INT_LIT
          | INT_LIT ':' INT_LIT ',' listadupla
          ;
```

e completá-las com as ações semânticas, vistas em aula, que informem as estruturas de dados internas que vocês optem por usar afim de representar o array na tabela de símbolos.

Cuidado: no `Pico`, quer-se poder declarar arrays cujos índices possam começar com um valor que não seja zero. As ações semânticas devem considerar esse caso.

Isso feito, deve-se passar a considerar na gramática a parte que possibilita os acessos a arrays:

```
lvalue: IDF
      | IDF '[' listaexpr ']'
      ;

listaexpr: expr
   | expr ',' listaexpr
   ;
```

Neste ponto, cada grupo possui duas opções para gerar o código TAC e calcular os atributos `codigo` dos não-terminais `lvalue` e `listaexpr`:

1. Manter a gramática entregue na etapa 2. Conforme explicado em aula, essa gramática não é compatível com o uso de apenas atributos sintetizados para calcular o código TAC, uma vez que o parser gerado pela Yacc reduzirá toda uma `listaexpr` antes de reduzir `IDF '[' listaexpr ']'`, e portanto não terá acesso ao `IDF` que indica o nome do array considerado, na hora de calcular o deslocamento pelo tratamento da `listaexpr`.

   Portanto, nessa opção, deve-se programar e chamar um procedimento que varrerá a sub-árvore de sintaxe reduzida pela `listaexpr` para nela calcular os atributos de código *depois* do parsing, da forma seguinte:

   ```
   lvalue: IDF
         | IDF '[' listaexpr ']'  { evaluate_code_array($3, $1);}
         ;
   ```

   onde `evaluate_code_array` pega por argumentos primeiro um ponteiro sobre a árvore (`$3` é a raiz da sub-árvore que representa toda a `listaexpr` usada para indexar o array. Observar que no ponto da chamada, essa árvore terá sido calculada pelo parser do Yacc); e segundo o lexema associado ao `IDF` que representa o array (`$1`). Este lexema poderá ser usado na implementação de `evaluate_code_array` para acessar a tabela de símbolos e nela recuperar as informações importantes para calcular o código TAC. Abaixo se encontra uma explicação um pouco mais detalhada de como funciona a avaliação recursiva dos atributos.

2. Alterar a gramática, para usar a segunda versão usada nos slides da aula de 25 de Outubro, que possibilita o uso de atributos apenas sintetizados. Neste caso, basta implementar o pseudo-código discutido nos slides da aula, como ações semânticas a ser executadas após cada redução pelo parser.

Avaliação Recursiva dos Atributos. Conforme foi explicado em aula, o percurso a ser implementado (na primeira opção acima) segue genericamente a forma seguinte:

Input: Um Node r, raiz de uma árvore de sintaxe, com campos att_herd e att_synt.
Output: A árvore com os atributos dos nós calculados.
`Avaliar` (**in-out** Node r) : AttType
$i \leftarrow 0$ `// Número do filho` n.
$s \leftarrow$ `New` Array of AttType `// Array de atributos.`
foreach $n \in r.children$ **do**
  `// Avalia os atributos de` n `que devem ser herdados de seu pai` r.
  n.att_herd $\leftarrow f(r)$
  `// Avalia (recursivamente) os atributos sintetizados de` n.
  $s(i) \leftarrow$ `Avaliar` (n)
  $i \leftarrow i + 1$
end
`// Calcula os atributos sintetizados de` r `a partir dos atributos de seus filhos.`
r.att_synt = g(s)
return $r.att\_synt$

Algoritmo 1: Cálculo dos atributos herdados e sintetizados.

No algoritmo 1, supõe-se que a estrutura de dados `Node` possui dois campos `att_herd` e `att_synt`, cada qual de tipo abstrato `AttType`. O primeiro contém o(s) atributo(s) herdado(s) pelo nó, o outro contém o(s) atributo(s) sintetizado(s). Nessa etapa, o primeiro campo não será usado. No algoritmo, a função `Avaliar` retorna o(s) atributo(s) sintetizado(s) do nó r passado em argumento. Supõe-se que o(s)

atributo(s) herdado(s) por um nó n de seu pai r são calculados por uma função $f(r)$[1], e que os atributos sintetizados de um nó n são calculados por uma função g aplicada aos atributos de seus filhos.

Na prática, você deverá implementar as funções f e g dependendo do tipo de nós na árvore de sintaxe. Por exemplo, se o nó r é um nó que representa uma soma aritmética, então ele deve possuir (entre outros) um atributo `code` sintetizado, que deverá ser calculado por uma função g que efetuará a concatenação dos atributos `code` dos filhos e de uma nova instrução TAC de soma.

Exemplo de código TAC esperado: a compilação de `z = A[i,j]; print(z)`, onde `A` seria um array bidimensional de inteiros, tendo 10 elementos na dimensão 1 e 20 na dimensão dois (por isso necessitando $10\times20\times$ 4 Bytes), onde `z` estaria mapeado no endereço 0, `i` no endereço 8, `j` no endereço 12 e `A` teria seu endereço base no endereço 16, resultaria na saída TAC seguinte:

```
816
16
000:   000(Rx) := 012(SP) MUL 10
001:   004(Rx) := 000(Rx) ADD 008(SP)
002:   008(Rx) := 004(Rx) MUL 4
003:   012(Rx) := 008(Rx)(016(SP))
004:   000(SP) := 012(Rx)
005:   PRINT 000(SP)
```

Nota-se neste exemplo a linha `003` que exemplifica o uso do operador de endereçamento indireto usado na linguagem TAC.

[1] Nota-se que, a rigor, f poderia pegar em argumento os "irmãos à esquerda" de n, além do pai r.

Chapter 5

Especificação da Etapa 5

Essa etapa final consiste na geração de código de três endereços ("TAC") para tratar o caso:

- do comando if... then... else;
- do comando while.

O executável `pico` será gerado a partir do código fonte e do Makefile do próprio grupo. Apenas o código TAC gerado por seu compilador será testado pelos scripts automatizados, o que lhes deixa bastante flexibilidade para programar as funcionalidades internas de seu compilador `Pico`.

5.1 Instruções TAC extras

LABEL.

`<label>:`

onde `<label>` é um identificador de rótulo válido. Um indicador de rótulo válido começa com uma letra (obrigatoriamente maiúcula) seguida ou não de qualquer combinação de letras maiúsculas, dígitos e *underscores*. Esse comando indica que a parte do código subseqüente está endereçada através de um comando `GOTO`. Exemplo de identificador de rótulo:

```
MEU_LABEL_2:
```

Atente que **não deve haver número da linha ou espacos na instrução**.

GOTO. A instrução `GOTO`:

```
xxx:    GOTO <label>
```

onde `xxx:` é um número de linha qualquer (como nas etapas anteriores, deve ser seguido de exatamente 3 espaços) e `<label>` é um rótulo definido como acima. Esse comando ajusta o fluxo de instruções para que passe a ser executado logo após (na linha seguinte a) um rótulo. *e.g.*,

```
000:    GOTO COMECA_AQUI
```

```
001:   004(Rx) := 000(Rx) ADD 008(SP)
COMECA_AQUI:
002:   008(Rx) := 006(Rx) ADD 4
```

nesse exemplo, a linha `001` nunca é executada.

Opcionalmente, o comando `GOTO` pode referenciar explicitamente uma linha no código TAC, dispensando a definição do rótulo. Nesse caso, o número da linha deve ser precedido de um *underscore*. O seguinte código faz o mesmo que o anterior:

```
000:   GOTO _002
001:   004(Rx) := 000(Rx) ADD 008(SP)
002:   008(Rx) := 006(Rx) ADD 4
```

(Observa-se que, através de um `GOTO _001`, seria possível executar a linha `001` afinal.)

IF. O comando `IF` é implementado como abaixo. Repare que o número de espaços entre os componentes deve ser respeitado (como sempre)!

```
xxx:   IF <op1> <comp> <op2> GOTO <label>
```

onde:

xxx é um número de linha qualquer.

op1 é um operando, que pode ser ou um número em valor absoluto ou um número armazenado na memória (como nas etapas anteriores, com referência aos registradores Rx e SP).

comp é um comparador booleano. São admitidos os seguintes comparadores:

- <
- >
- <=
- >=
- ==
- !=

op2 é como op1.

label é como o rótulo definido para a instrução GOTO (aceitando, inclusive, o número da linha, precedido por um *underscore*, conforme explicado anteriormente).

e.g.,

```
000:   IF 4 <= 008(SP) GOTO MEU_LABEL
001:   004(Rx) := 000(Rx) ADD 008(SP)
MEU_LABEL:
002:   008(Rx) := 006(Rx) ADD 4
```

executa a linha `001` apenas quando a comparação da linha `000` resultar em falso.

5.2 Cálculo dos Atributos Semânticos

Nessa etapa, precisa-se usar atributos sintetizados e herdados. Seu cálculo implicará em percorrer a árvore de sintaxe em profundidade. No capítulo anterior consta uma (re)explicação a respeito da programação deste cálculo (p. 38).

5.3 Trabalho exigido

Vocês devem implementar o tratamento dos comandos `if... then... else` e `while...`, conforme foi apresentado em aula.

A avaliação das expressões booleanas, necessárias para o controle das instruções `if` e `while`, deverá ser feita com código do tipo "curto-circuito" e levar ao valor correto dos atributos `falso` e `verda` dessas expressões. Para isso, você irá precisar de um novo procedimento chamado `gera_rotulo()` que irá gerar a cada chamada um novo rótulo simbólico; um tal rótulo será uma seqüência de caracteres quaisquer, seguidos por dois pontos (`:`). Por exemplo, `gera_rotulo` pode retornar sucessivamente: `label0:`, `label1:`, `label2:`, etc...

Para as ações semânticas necessárias, ver a aula do 03 de novembro.

Seu parser chamado `pico` deve aceitar como argumento na linha de comando:

- o nome do arquivo contendo o input (já era o comportamento do seu scanner). O arquivo de entrada deve obrigatoriamente ser sufixado pela extensão `.pico`. Este argumento deve ser o último da linha de comando.
- o flag `-o`, seguido pelo nome do arquivo de saída. Este nome será sempre sufixado pela extensão `.tac`. O código TAC gerado pelo compilador deve ser escrito neste arquivo (se a compilação for bem sucedida).

Exemplos de uso:

```
pico -o test1.tac test1.pico
pico -o output.tac input.pico
pico -o resultado.tac prog_1_.pico
```

Para testar se seu código TAC é correto, poderá ser usado o script que converte um programa TAC para assembly x86 (tradutor) e o executa.

Qualquer esforço extra além do trabalho exigido será levado em conta para a definição da nota de apresentação final (por exemplo: boas mensagens de erro; checagem de limites nos acessos nos arrays; ....).

Chapter 6

Anexo técnico: Doxygen e Make

6.1 Introdução ao DOXYGEN

Todo o conteúdo deste tutorial foi retirado de http://www.stack.nl/~dimitri/doxygen/manual.html. Recomenda-se consultar a fonte para mais detalhes. O que segue são as notações recomendadas para o projeto da disciplina de Compiladores (INF01147/INF01033) 2008/2.

Gerador de documentação para código-fonte, o DOXYGEN é um sistema de documentação para C++, C, Java, Objective-C, Python, *etc.*

Dentre outras capacidades, o DOXYGEN pode gerar um sistema de documentação *on-line* (em HTML) e/ou um manual de referência *off-line* (em LaTeX) de um conjunto de códigos-fonte propriamente documentados. Também há suporte para gerar a saída em RTF (*MS-Word*™), *PostScript*, PDF com *hyperlinks*, HTML compactado e páginas *man* do UNIX. A documentação é extraída diretamente dos códigos-fonte, tornando mais fácil a manutenção da documentação. Disponível para Windows™, Linux e MacOS-X™.

6.1.1 Comandos Básicos em C

Existem muitos métodos diferentes para escrever um comentário no estilo DOXYGEN . Listados, a seguir, os mais comuns.

6.1.1.1 Blocos de Comentários Simples

Comentários simples na sintaxe da linguagem podem ser colocados normalmente. Por padrão, o DOXYGEN **não** gerará estes comentários na saída, ignorando sua presença. São necessários identificadores especiais para os comandos DOXYGEN .

```
/* comentario simples */
```

6.1.1.2 Blocos de Comentários Breves

Um dos modos em que o DOXYGEN pode gerar comentários, estes são produzidos na saída; devem possuir apenas uma linha, são usadas para considerações pontuais importantes (como a descrição do que faz um laço `for` complexo) e sua sintaxe é a seguinte:

```
/** comentario breve */
```

Também pode ser obtido através da *tag* `\brief`:

```
/* \brief comentario breve */
```

6.1.1.3 Blocos de Comentários Detalhados

Um dos modos em que o DOXYGEN pode gerar comentários, estes são produzidos na saída; usados para descrições detalhadas de blocos dos arquivos, em várias linhas (como o *header* do código fonte ou cabeçalho de função). Sua sintaxe é a seguinte:

```
/**
 * comentario detalhado
 *
 */
```

6.1.1.4 Tags Especiais

Algumas *tags* (marcações especiais do código que aparecem an documentação final) são de especial uso neste projeto:

author : nome autor do autor do código em questão (vários autores de vários `author` serão agrupados no mesmo parágrafo).

version : versão atual do código (*e.g.*, número da etapa do projeto).

date : data da primeira e última modificação no código.

bug : descreve possíveis *bugs* que a aquele trecho de código apresenta.

warning : aviso no uso daquele trecho de código.

param : identifica o parâmetro de uma função. Deve ser usado com as palavras-chave *in* e *out* para indicar o tipo de atributo (entrada ou saída) e uma breve descrição.

return : valor de retorno de uma função.

file : identifica o arquivo (nome dele no sistema de arquivos) ao qual o comentário se refere. Em geral é o nome do arquivo de código-fonte e aparece apenas no *header*.

Por exemplo,

```
/**
 * \file minhas_funcoes.c
 * \brief Arquivo com as minhas funcoes.
 */

/** MinhaFuncao
 *  \brief Copia sequencias de bytes.
 *  \author Bill Gates
 *  \author Linus Torvalds
 *  \version 4.0
 *  \date    1996-1998
 *  \bug Trava muito e requer muita memoria.
 *  \bug A medida que eh usada introduz mais bugs.
```

```
 *  \warning Nao confie nesta funcao.
 *  \warning Nem neste comentario.
 *
 * Nao faz nada e soh consome memoria.
 * \param[out] dest Area de destino.
 * \param[in]  src  Area de origem.
 * \param[in]  n    Numero de bytes.
 * \return Nada.
 */
void minha_funcao(void *dest, const void *src, size_t n);
```

6.1.1.5 Uso

O DOXYGEN consegue determinar a que bloco de código um comentário se refere. Isso significa que blocos que antecedem uma função referem-se à função e blocos que antecedem todo o arquivo são o *header* do arquivo na documentação. Neste projeto, três blocos são o suficiente para que o arquivo seja considerado documentado: o *header*, o descritor da função e comentários de trechos importantes de código. Em anexo, segue um exemplo.

6.2 Introdução ao Make

6.2.1 Introdução

Make é um programa GNU linux que automatiza o procedimento de criação e manutenção de grupos de programas, verificando dependências e possibilitando a compilação de códigos. O objetivo de make é determinar automaticamente as partes de um programa relativamente grande que precisam ser recompiladas, provendo os comandos necessários para tal finalidade. Com make podem ser descritas muitas tarefas onde arquivos precisam ser atualizados automaticamente a partir da modificação de outros.

Em um programa, o arquivo executável é obtido a partir de arquivos objeto, os quais são oriundos da compilação de arquivos fontes. Com apenas uma única chamada de make, todas as mudanças realizadas em arquivos fonte podem ser recompiladas, uma vez que make se baseia nos tempos das últimas modificações do arquivo. Caso não haja nenhuma alteração, não será necessária a compilação, não havendo a necessidade de recompilação de todo código

6.2.2 Composição de um arquivo makefile

Para usar make, é necessário criar um arquivo chamado *makefile* que descreve as relações entre os arquivos do programa e os estados dos comandos para a atualização de cada arquivo. Em um arquivo makefile podem ser utilizados *labels* (rótulos) e *targets* (alvos), de maneira que na chamada de make possa haver o reaproveitamento de nomes, bem como a invocação de diferentes alvos. Também é possível verificar dependências. Por exemplo, se o arquivo makefile tiver a seguinte especificação:

```
FILENAME=fonte
FLAGS= -g -Wall

compilar:
        ${CC} -o ${FILENAME} ${FILENAME}.c ${FLAGS}

executar: ${FILENAME}
        ./${FILENAME}
```

```
apagar:
        rm ${FILENAME}.o
        rm ${FILENAME}
clear
```

Tem-se um *label*, $FILENAME$, o qual está referenciando a palavra "fonte", sendo usado em diferentes partes do script como uma "variável" e a *label* FLAGS, o qual é invocada apenas em uma das *targets*

Ao mesmo tempo, têm-se três *targets* que podem ser chamados através de:

```
nome@maquina:/home/usuario: make compilar

nome@maquina:/home/usuario: make executar

nome@maquina:/home/usuario: make apagar
```

No primeiro caso haverá a compilação do código utilizando o compilador gcc. Este é invocado através de uma variável externa. Já as *flags* de documentação estão referenciadas em *FLAGS*.

No segundo caso a *target* verifica a existência do código executável (repare que isto é feito logo após os dois pontos) e se isto for verdadeiro ele executa o código.

Já no terceiro caso haverá a execução de três comandos: a exclusão do arquivo objeto, a exclusão do arquivo executável e a "limpeza" do terminal.

Para o acesso ao conteúdo de um *label* basta utilizar $ antes do *label*, sendo esta colocada entre chaves. Outra observação importante é de que os comandos precisam ser colocados a partir da linha seguinte de *target*, iniciando sempre com um *tab*.

É através da invocação dos targets que se pode executar uma série de comandos práticos, simplificando o processo de compilação.

6.2.3 Maiores Referências

Sugere-se a consulta de:

$man\ make$

$http://www.gnu.org/software/make$

$http://www.gnu.org/software/make/manual/make.html$